AFRL-IF-RS-TR-1999-54
Final Technical Report
March 1999

A GENERIC MANAGEMENT AND CONTROL API BASED ON AN OBJECT-LEVEL INTERACTION PARADIGM AND ITS DEMONSTRATION IN A VIRTUAL WORKSHOP SERVICE

Columbia University

Aurel L. Lazar, Rolf Stadler, Constantin Adam, Cristina Aurrecoechea, Marco Borla, Mun Choon Chan, Andreas Eggenberger, Jean-Francois Huard, Koon Seng Lim, Mahesan Nandikesan, Roman Puttkammer

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

APPROVED: [illegible]
BRADLEY J. HARNISH
Project Engineer

FOR THE DIRECTOR: [illegible]
WARREN H. DEBANY, Jr., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1999	Final Sep 95 - Feb 98

4. TITLE AND SUBTITLE
A GENERIC MANAGEMENT AND CONTROL API BASED ON AN OBJECT-LEVEL INTERACTION PARADIGM AND ITS DEMONSTRATION IN A VIRTUAL WORKSHOP SERVICE

5. FUNDING NUMBERS
C - F30602-95-C-0015
PE - 62702F
PR - 4519
TA - 22
WU - 39

6. AUTHOR(S)
Aurel L. Lazar, Rolf Stadler, Constantin Adam, Cristina Aurrecoechea, Marco Borla, Mun Choon Chan, Andreas Eggenberger, Jean-Francois Huard, Koon Seng Lim, Mahesan Nandikesan, Roman Puttkammer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Center for Telecommunications Research
801 Schapiro Research Building
Columbia University
New York, NY 10027-6699

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL/IFGA
525 Brooks Rd.
Rome, NY 13441-4504

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-1999-54

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: Bradley J. Harnish, IFGA, 315-330-1884

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
The goal of this work was to define generic application programming interfaces (APIs) for the control and management of an ATM-based telecom environment. In our approach the APIs are defined, based on an object-level interaction paradigm, as object interfaces. We define a broadband kernel as a CORBA-based distributed programming environment that facilitates the easy creation of network services and provides mechanisms for resource allocation. The primary service provided by the broadband kernel is one-to-N one-way end-to-end connectivity with quality of service. More complex services are generated as aggregations of this and other primary services. A virtual workshop service was designed and implemented to experiment with the control and management APIs. This report puts together the building blocks of a telecom architecture: the Binding Interface Base models the resources, and the set of broadband kernel algorithms are modeled as a set of interacting objects offering their APIs to higher level services and applications. This report also presents our view on management. A model for management was developed that explains how this set of interacting objects and the services provided are managed in a consistent manner.

14. SUBJECT TERMS
application programming interfaces, object interfaces, object-level interaction, broadband kernel, CORBA, distributed programming environment, quality of service, virtual workshop, telecom architecture, Binding Interface Base, service management

15. NUMBER OF PAGES
202

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Table of Contents

Table of Contents

1 Introduction

The goal of this work is to define *generic* APIs for the control and management of an ATM-based telecom environment. In our approach the APIs are defined, based on an Object-Level Interaction Paradigm, as object interfaces. A Virtual Workshop service is designed and implemented to experiment with the control and management APIs.

First a summary of the initial proposal statement is given. Next we introduce the work presented in this report, that is, the conceptual framework XRM/RGB and in it the main system components of our Network and Service architectures.

1.1 Initial Work Proposal

Broadband networks make it possible to deploy a wide range of distributed applications that require quality-of-service (QOS) guarantees. Different distributed systems cooperate in order to provide these QOS guarantees to distributed applications; examples of such systems are connection management, real-time control, and network management.

The proposed work is to develop a *framework for interaction and cooperation* between distributed applications and the broadband network systems that provide communication, control and management capabilities for them. We model these interactions as a set of interfaces between the different components of the system. Hence, the purpose of these interfaces is to allow service creation, invocation, control, and management.

Figure 1 defines the main functional components of a distributed application and the broadband network that supports it.The broadband network is organized into information transport, network control and network management functionality. This subdivision is based on the principle of separation between controls and communications and reflects the different time-scales on which these subsystems operate (control on the fast time scale and management on the slow time scale). The information transport system models the (ATM) protocols and entities for the transport and processing (switches, links, computing platforms, etc.) of user information.

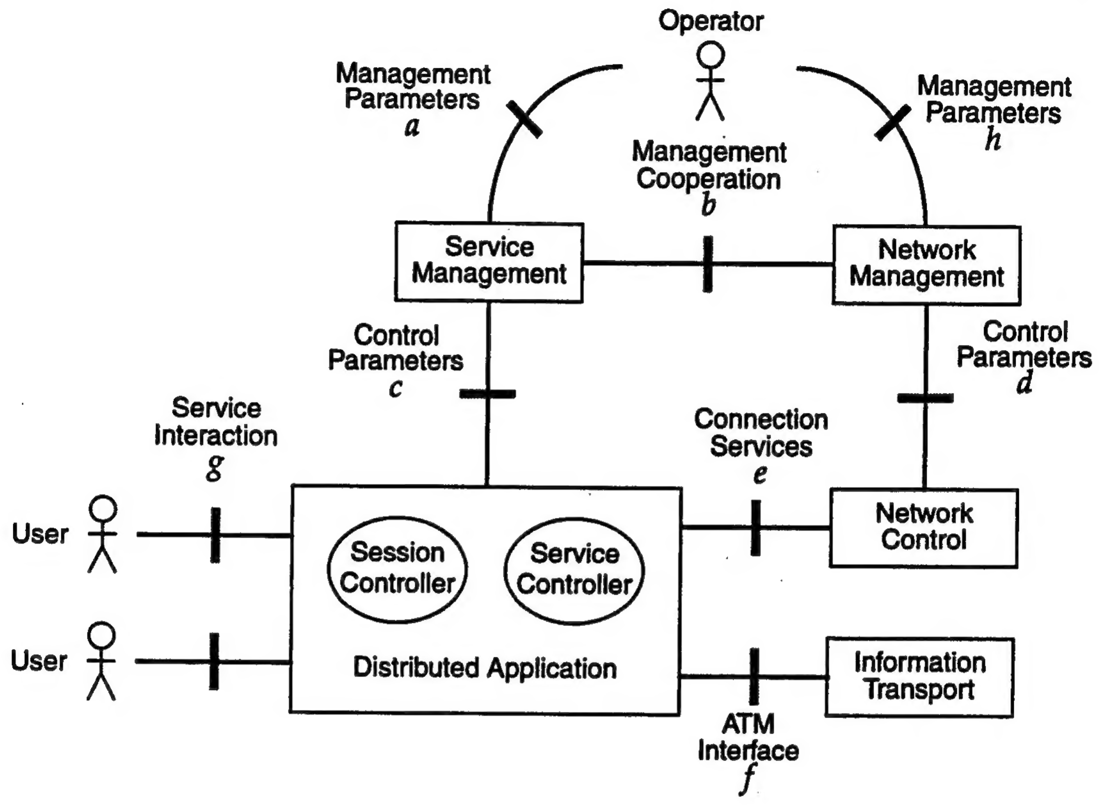


Figure 1 Cooperating systems and interfaces to support distributed applications on broadband networks

The *network control system* incorporates the resource control and the connection management subsystems. *Connection services*, provided by the connection management subsystem, establish, maintain, and release end-to-end connections. These services differ in the type of connections they employ and in the style of interactions that take place among functional entities, e.g., synchronous or asynchronous [18]. (A connection is a unidirectional communication channel between application processes at different locations; it can be either unicast or multicast. A connection has a set of QOS constraints associated with it.)

The resource control subsystem can be seen as a collection of mechanisms, each of which operates asynchronously and solves a specific resource control problem. Examples of real-time control mechanisms are buffer management and scheduling, flow control, routing and admission control. The operations of the resource control subsystem can be tuned by changing control parameters associated with each mechanism.

The management system is controlled by a human operator. Two management systems are identified -- service management and network management -- and these differ in their view of the system and the granularity of their objectives[20].

The *network management system* interacts with the real-time control system, following the monitor/control paradigm. This means that it monitors the network state and takes control actions in order to influence this state. Control actions result in changing specific parameters in the real-time control system. The interaction of the performance management system with the real-time control system is asynchronous, due to the different time scales on which the functional components in both systems run.

Network operators perform actions to influence the network state, and are responsible for achieving high-level management objectives. They monitor the network state represented as visual abstractions on a graphical interface, and perform operations by acting upon management parameters [19].

The architecture in Fig. 1 shows the functionality of the *distributed application* split into two separate entities: service controller and (user) session controller. The session controller is associated with each user of the distributed application and provides the necessary access functionality. The service controller provides a holistic view of the distributed application. It gathers the computing and communications resources needed by the distributed application and interacts with the network control system, in order to set up and maintain the communication resources required to support the distributed application.

The *service management system* interacts with the service control system by changing specific parameters. The interaction is asynchronous due to the different time scales on which the functional components in both systems run. The operator performs actions to achieve service management objectives. The primary objective is to guarantee that each user receives a satisfactory level of service.

Since new multimedia applications will demand different types of connection services, the capability to support new connection services by the network control system is a fundamental requirement for future broadband networks [15]. Future broadband networks will provide higher level communication services, called teleservices in [11], using functionality offered by lower level services. Therefore, we envision a *service creation* environment where a service can be easily deployed by composing existing services and combining them in a specific way.

As explained in more detail below, we propose a new paradigm for interaction among application, control and management systems that allows the handling of connections with guaranteed end-to-end QOS requirements, while allowing for transparent integration of new services. Our paradigm is based upon the concept of *object-level interaction*, which implies that the interface of a system is modeled and implemented as a set of objects that can be accessed from outside.

Our focus is on the interaction between the systems described in Fig. 1. We propose to design and implement interfaces using the concept of *object-level interaction*. This will allow for the construction of interfaces that are generic, in the sense that they apply to classes of services and applications, not just to specific instances. The proposed paradigm will further lead to extensible and flexible interfaces, i.e., to interfaces that can frequently be enhanced to support new functionality and new services. Since all systems cooperating in the task of service creation and delivery are distributed, it is important that their interfaces can be changed without requiring recompilation and restart of applications or control systems. Our implementation will support this feature.

Our interaction paradigm will allow for dynamic renegotiation of resources and QOS parameters. This is necessary because resource requirements of an application or a service may change over time, and because the network control system may need to redistribute the available resources among the services currently active, in order to meet the requirements of efficiency and QOS guarantees. Our paradigm for interaction will be applied to support performance objectives of the cooperating systems (viz., the service providers and the network provider) and the users.

The architecture introduced in Fig. 1 encompasses a single network domain. This means that we assume that the network management system has control and monitoring authority over all subsystems in a particular domain. However, we will study an extension of our work in a *multidomain environment*, by developing generic interfaces to a peer-to-peer management system and a traffic control system. Fig. 2 shows how our architecture can be extended to include interaction among multiple network domains. The architecture in Fig. 2 will allow for a scalable system that extends over multiple administrative regions (e.g., the system can extend over a public network) or heterogeneous platforms.

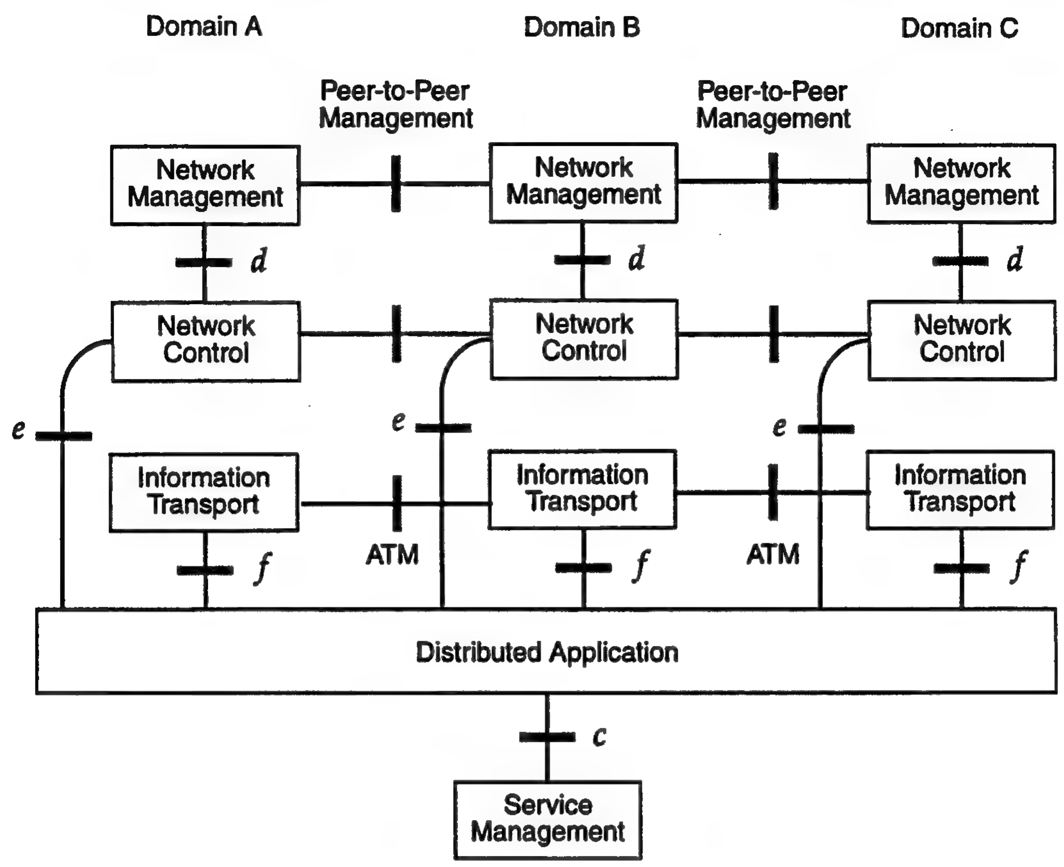


Figure 2 Cooperating systems and interfaces in a multidomain environment

In order to demonstrate the capability of our object-level interaction paradigm, we propose to design and implement a *Virtual Workshop* (VW) service, which is a distributed multimedia application that provides various forms of multimedia associations among VW participants. As will be described in the next section, the VW is a complex service that will easily allow us to experiment with our interface designs under a variety of demanding service requirements.

We propose to design and implement the architecture described in Fig.1 on a broadband network platform that is being deployed at Columbia University and that will be extended into NYNET, a New York State broadband network which interconnects Universities and Research Laboratories in the New York State and Massachusetts areas.

1.1.1 Modeling Generic Interfaces

Our approach for devising generic control and management interfaces is based upon the concept of *object-level interaction* between different systems. This means that the interface of a system is modeled and implemented as a set of objects that can be accessed from outside. Understanding an interface as a set of objects makes it both *modular* and *extensible*: modular, because it consists of a *set* of entities, and extensible, because extending the interface translates into adding objects to the system. A specific object in the interface of each system can be accessed to get information about the current set of services offered at this particular interface. Our approach contrasts with the traditional view of interfaces which is based on protocol interaction and promotes the view of a static and monolithic access point with an unstructured set of predefined services.

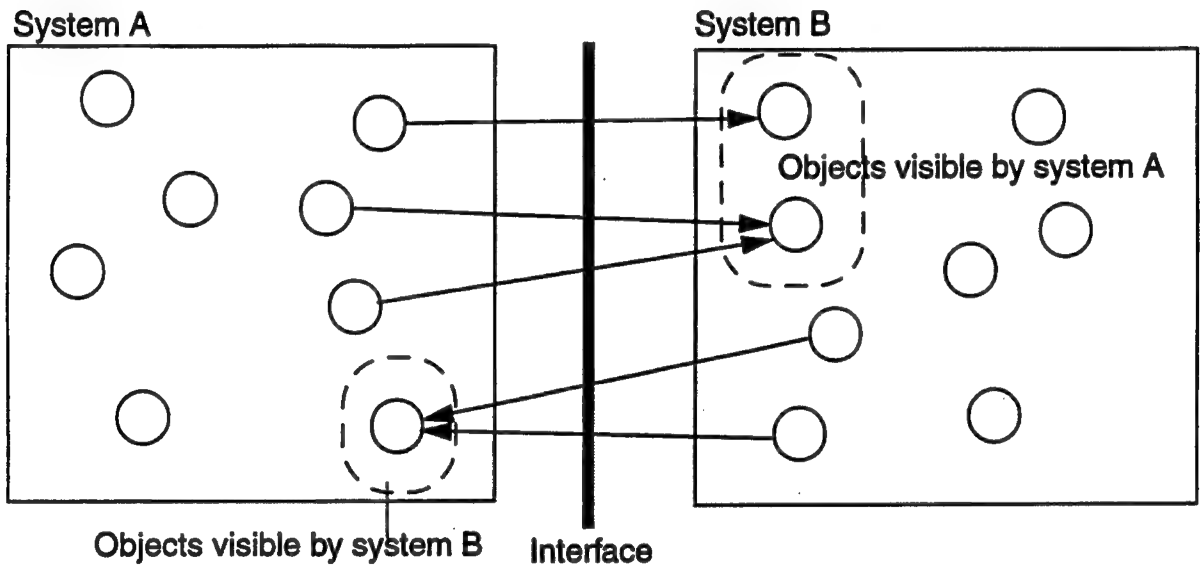


Figure 3 Creating flexible and extensible interfaces using object-level integration.

Our choice is to use an object-oriented (OO) distributed system development environment for the design of these generic interfaces. First, the complexity of the system makes it a primary candidate for using OO based software engineering techniques for its design; second, the nature of a telecommunication system is a distributed one in which basic components need to interwork seamlessly in geographically dispersed locations. This OO "middleware" provides the transparency for achieving the necessary interoperability and flexibility. Scalability of this architecture is achieved with a distributed object-oriented design.

We apply OO techniques to model applications and all cooperating systems as sets of objects interacting with each other. The interface between two systems is determined by the interaction of objects belonging to different systems.

Our methodology for service design [3] is based on the Extended Reference Model (XRM) [14], which defines the need for four types of interfaces that correspond to the separate functional entities introduced in Fig. 1. Our methodology for creating generic interfaces will be based on an inheritance hierarchy, with a root object that is an aggregation of a connection management, a resource management, a service management, and a transport interface object. A generic interface thus consists of a set of objects the number of which can vary. This implies that by adding new objects to a system interface, its instantiation might become any of the four classes above or a clustering of all four. This, of course, allows us both modularity and extensibility in the interface design and instantiation.

The object interfaces are conceptualized as a common distributed data repository called the *Binding Interface Base (BIB)* [16].

Based on our experience in the definition of interfaces [3][4], we propose the interface structure shown in Fig. 4 for specifying generic interfaces. The structure clarifies the dynamics in the object interactions that take place in a generic interface. Indeed, an operation in an object interface can be (1) a factory operation, (2) a service operation, or (3) a management operation. Service operations can be further classified into three types: (a) query operations, (b) modify operations and (c) notification operations. While factory operations allow the creation of new objects, management operations act upon the attributes that characterize each object. We consider three types of attributes: (1) state, (2) environmental, and (3) behavioral attributes. The type state is consistent with the OSI State Management function [12]. The environmental attributes contain the information needed by an object to perform its factory and service operations. An example of an environmental attribute is the interface of an object to which a notification has to be sent, or the topology of a domain. Finally, behavioral attributes affect the execution of the factory and service operations by modifying the sequence of actions to be performed.

```
interface generic {

// ATTRIBUTES LIST:
// - state attributes
// - environmental attributes
// - behavioral attributes

// OPERATIONS LIST at the interface:
// - factory operations:
// - modify operations:
// - query operations:
// - notification operations:
// - management operations: get/set upon the attributes

// For each operation specify:
// - preconditions
// - sequence of actions
// - postconditions
}
```

Figure 4 Interface Structure

1.1.2 The Virtual Workshop service

In order to demonstrate the capability of our object-level interaction paradigm, we will develop an advanced multimedia service that will require interaction among the various entities described in Fig. 1. Specifically, we propose to design and implement a *Virtual Workshop* service. As already mentioned, the Virtual Workshop service is a distributed multimedia application that provides various forms of multimedia associations among VW participants. Like a "real workshop" a Virtual Workshop features interactions such as sessions and tutorials, as well as one-to-one conversations among participants. In contrast to a "real workshop" however, the VW participants are physically distributed instead of meeting in the same physical location. They make use of the broadband communication infrastructure and the VW service in order to interact.

As shown in Fig. 5, a VW appears as a collection of parallel sessions of different kinds, such as, tutorials, birds-of-a-feather sessions, workshop sessions, and others. The "Session Facto-

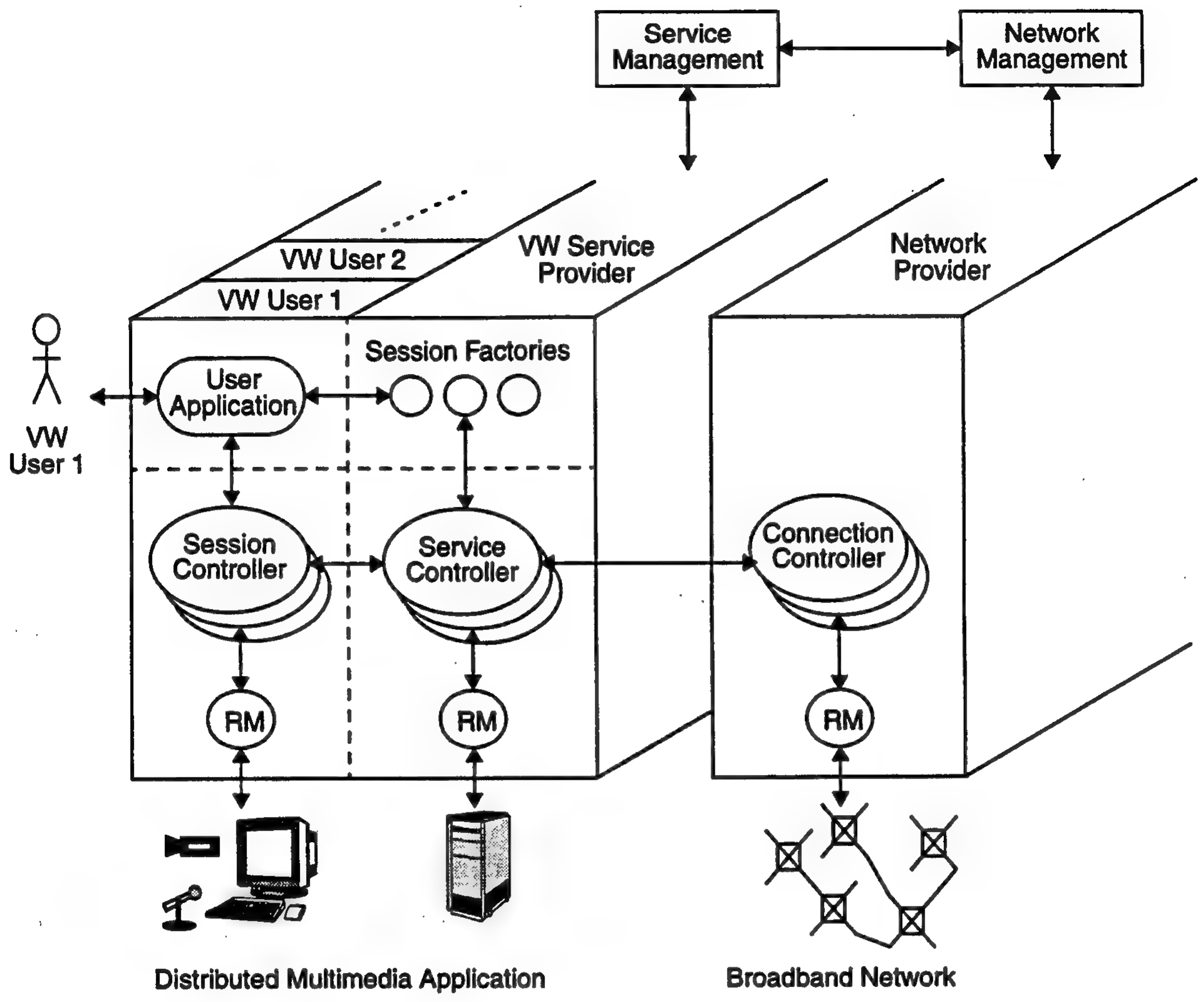


Figure 5 Cooperating systems in the virtual workshop application

ries" represent objects that initialize a new session upon request and allocate the necessary resources for that purpose. After admission to the VW, a participant can establish, create, and instantiate new sessions as well as join or leave existing sessions. These requests generally involve interactions between the distributed VW application and the network control system.

VW participants can take on different roles. In an interactive session, for example, the user in the role of the chairman regulates the interactive discussion (i.e., he/she executes floor control), timing the speakers' presentations and the questions/answer periods. Participants in the role of attendants listen to the various presentations and can ask questions as scheduled by the chairman. Participants can "flip" between several parallel sessions visible on their workstations, engage in a one-to-one communication with other participants, or even suspend and later resume sessions to perform some high-priority work, such as answering an important electronic mail.

The Virtual Workshop service is a convenient ground to experiment with distributed multimedia applications interacting with the network and the management system. It combines distributed applications with different requirements, such as information retrieval type of services, message type of services, or teleconferencing services. Most importantly, such a VW service allows us to demonstrate the fundamental aspects of our interface approach:

- First, the object interfaces for network services, such as a unicast or multicast connection service, will be refinements of the generic network service interface, which, in turn, will be a refinement of the generic service interface. This shows the inheritance property of generic interfaces based on an OO design methodology.
- Second, during the course of the project, the functionality of the different components of the system will evolve. So will the services provided by the interfaces. This allows us to prove the aspect of flexibility and extensibility of the paradigm of object-level interaction as introduced above.
- Third, we will be able to demonstrate the functionality of the interaction between the different systems shown in Fig. 5, while the involved entities -- VW participants, the VW service provider, and the network provider -- attempt to meet their respective performance objectives. In Sec. 1.2 we will describe the four different scenarios that are envisioned for demonstrating the functionality of the proposed interfaces.

From the point of view of the network operator, the VW service appears as a graph of (inter) connections between various sources and sinks. Parallel sessions are seen as unicast and multicast connections associated with the specific VW and are set up through the interconnection graph. The network management system allocates bandwidth to each of the connections, thus assigning communication resources to a VW. During the operation of the VW, the need for communication resources may change dynamically. Therefore, the service management system can renegotiate with the network management system the bandwidth available to the VW. This negotiation is driven by the objectives of the service management system to satisfy the user requirements while minimizing the cost of network resources.

The view provided by the network management system to the operator includes network services and resources allocated to them. Specifically, the operator sees all connections and the associated resources that are currently active in the network, as well as their association to specific services. On the other hand, the view provided by the service management system to the operator includes the services that comprise a particular VW. It shows, for example, the participants of a VW, their roles, and the network connections (unicast and multicast) that are currently active in this particular VW.

1.1.3 Implementation Approach

The choice for our OO development environment is a CORBA compliant system, because of its wide acceptance as an industry standard [8] and also because of the rich functionality it provides [9][10][13].

A CORBA system is platform independent. This characteristic allows us to use CORBA for developing both the network platform as well as the service support on the computing platform. The use of CORBA in both domains provide us with an elegant solution to the problem of interworking distributed objects, without confining us to use a specific platform.

Another type of unification we achieve by using CORBA is the possibility of using it for both control and management activities. Control and management systems share information and differ in the time scale of their operations and in the level of distribution decided upon for their system architecture. We propose to use CORBA for both activities as a simple implementation solution to their need to share information.

CORBA features dynamic interface invocation which allows new types of objects to be added and accessed during run-time. This enables us to realize modular yet dynamic interfaces which have the flexibility of being changed over time. This added flexibility gives us an edge over traditional programming paradigms because new services can be implemented and integrated 'on-the-fly' without the need for a system-wide recompilation or even disruption to currently running services.

xbind [4] is a CORBA based implementation of the Binding Architecture. In **xbind,** object interfaces are specified using the CORBA Interface Definition Language (IDL). The status of the Binding Interface Base (BIB) is described in [4]. Implementation experience has shown the need to add structure to the CORBA IDL generic object specification. The structure we choose is shown in the appendix. We believe it helps in specifying behavior and therefore understanding dynamic interactions -- a specification that is not covered by CORBA IDL.

We propose to use **xbind** as a platform for implementing the generic interfaces described in the previous sections of this project. The **xbind** system already provides us with the fundamental underlying support services necessary in a multimedia network with QOS guarantees. The **xbind** implementation has been tested and its architecture is consistent with our proposed interface design.

Although our design and implementation focuses on a single domain, note that the **xbind** architecture and implementation platform allows an extension into multiple domains, thus including interaction with a peer-to-peer management system. This can easily be achieved via proxies that translate CORBA into specific GDMO or UNI signalling type of interactions. In [17] it is described how ATM Forum UNI (Version 3.0) can be implemented on top of the binding architecture for multimedia networking. In **xbind** terminology, the UNI is a binding algorithm that uses the interfaces offered by the data abstractions in the binding interface base. Interaction with OSI management systems can be solved in a similar way, where OSI managed objects are specified using GDMO.

1.2 Accomplished Work

The Extended Reference Model (XRM) [LAZ92] models the communication architecture of networking and multimedia computing platforms. It consists of three components called, the Broadband Network (the R-model), the Multimedia Network (the G-model), and the Services and Applications Network (the B-Model, see Figure 6). XRM facilitates the task of building telecom multimedia services; its power and functionality can be characterized by the power and variety of APIs it offers, so they can be used under different economic policies and business practices. XRM structures its APIs under two kinds: QoS abstractions located at the boundary R || G and service abstractions located at the boundary G || B.

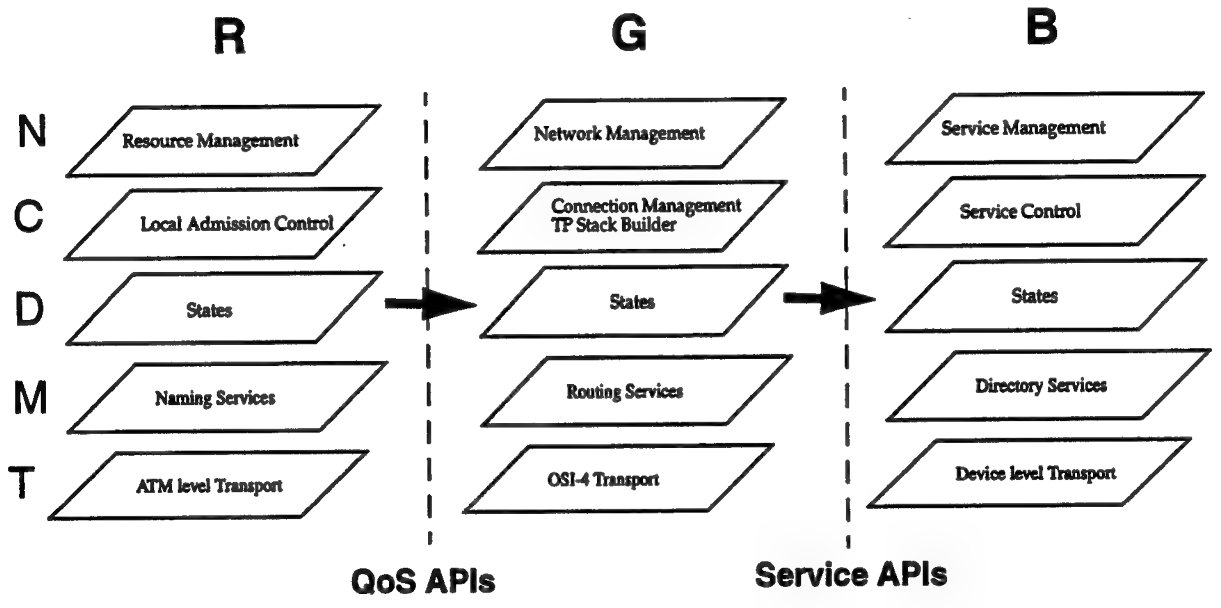


Figure 6 XRM

In this context, we define the *Broadband Kernel* as a CORBA-based distributed programming environment that facilitates the easy creation of network services and provides mechanisms for resource allocation. The primary service provided by the Broadband Kernel is 1-N oneway end-to-end connectivity with quality of service. More complex services, such as a Virtual Workshop, are generated as aggregations of these primary services.

The Broadband Kernel consists of a collection of organized interfaces called the Binding Interface Base (BIB), and a set of algorithms that operate on these.

The BIB abstracts the capacity and state of individual network and end-system multimedia resources and provides APIs to these abstractions. The abstractions are local in nature in that they do not capture any distributed interaction. Some of the BIB interfaces are collected together in container objects, such as the `SwitchServer` and the `AdapterServer`.

As said before, the primary service provided by the Broadband Kernel is 1-N oneway end-to-end connectivity with quality of service. This entails binding (interconnecting) BIB objects to create the communication service. This is accomplished via a set of algorithms such as routing, connection management and transport, all part of the Broadband Kernel. Figure 7 explains how some of these algorithms (state distribution, routing and connection management) interact.

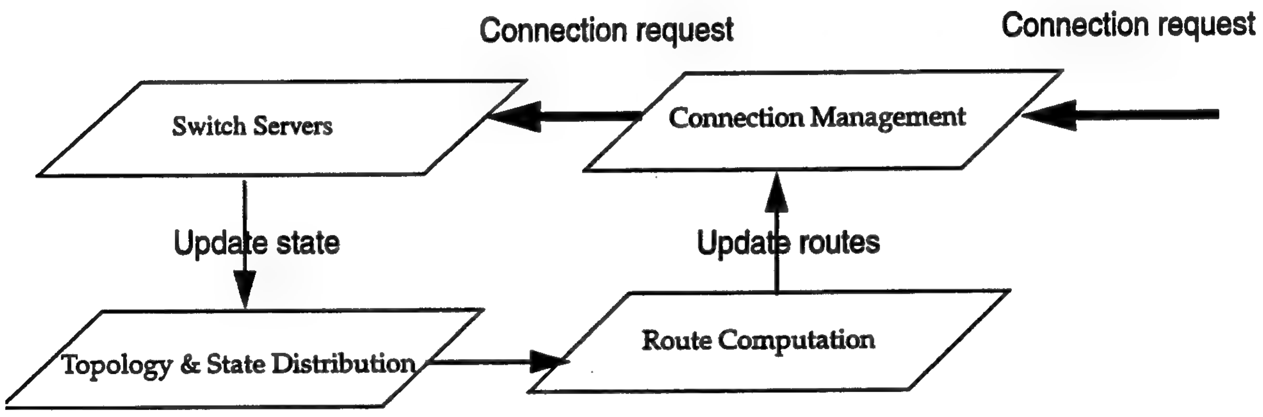


Figure 7 Interaction between tasks running at different time scales

The connection management service is responsible for binding the network resources to create connections; this activity is represented in Figure 7 by thick arrows. Routing provides connection managers with routes. Since route computations could be time-consuming, the routing algorithms run asynchronously with the connection management service; the routing activity is represented in Figure 7 by thin black arrows. Finally the topology and state distribution service collects from the switch servers the state and capacity of the individual network resources, such as schedulable regions, to be used by the route computation algorithms.

This report puts together the building blocks of a telecom architecture: the BIB modeling the resources and the set of Broadband Kernel algorithms, modeled as a set of interacting objects offering their APIs. The BIB is presented in the Appendix; the set of Broadband Kernel APIs are introduced next.

Finally, to exemplify the service creation process this report describes a service example: the Virtual Workshop. Also, this report presents our view on management: a model for management has been developed that explains how this set of interacting objects and the services provided are managed in a consistent manner.

1.2.1 Network Switches and Adapters

Resource Control is accomplished via two main APIs described in Chapter 2: the AdapterServer API and the SwitchServer API. Sometimes the term NodeSever (mostly under Connection Management) will be used to refer to both objects. Both objects are containers for a number of other objects which represent the network resources, included in the BIB described in the Appendix.

1.2.2 Routing Model

Routing specifies three main APIs: the Networking Capacity Graph (NCG) object is a database for the network topology, schedulable regions and other link-state information; the Route Repository (RR) is a database for storing routes; and the Route Computation (RC) object which precomputes the routes based on the current NCG, following a specific algorithm.

Figure 8 shows a view of the routing system based on time scale decomposition, which can be mapped to the XRM. The D-plane or Data plane of the XRM contains the states. With reference to routing, it contains the NCG and RR objects. The M- and C- planes consist of control algorithms that operate on D-plane objects. The M-plane algorithms are in charge of resource allocation - in the present case, routing. The C-plane algorithms are in charge of resource binding. The separation between the M- and C- planes may also be viewed from the point of time scales. The connection management service runs at the time scale of call arrivals and departures whereas the routing algorithms run asynchronously at their own rate.

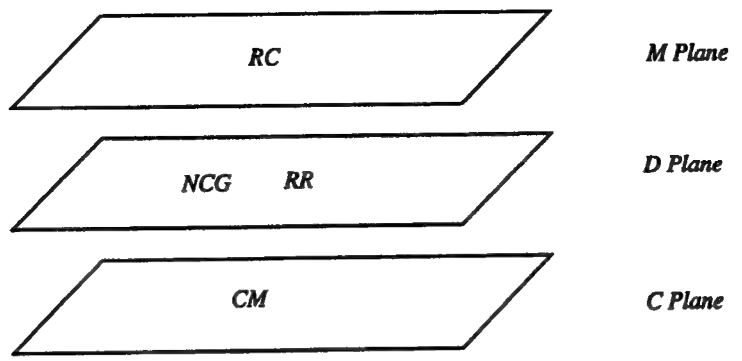


Figure 8 Relationship of the routing architecture to the XRM.

1.2.3 Connection Management

Connection Management is in charge of binding network resources to provide end to end connections. An object called connection manager (CM) will receive the request to initiate the binding. The binding will be accomplished via requests to individual network switches and adapters, once the route has been decided. As part of this project a specific connection management algorithm has been designed: *mcast*.

mcast is a specific ATM multicast service based on an object oriented design. It provides the capability of transporting multimedia information from a source to one--or more than one--destination. An *mcast* session belongs to one specific service class, i.e., video, data or voice. (A service class defines the performance characteristics and the performance requirements of a cell flow.) An *mcast* session can be represented as a graph, a tree where the root is the source of information and the leaves are the sinks (aka. receivers, destinations)[1].

mcast design includes two APIs: the *mcast* directory service (MDS), and the *mcast* connection manager (MCM). No controller has a global view of the ongoing sessions; knowledge is truly distributed.

1. A session can exist without destinations.

On request the MCM will access the NodeServer API which realizes the network resources. This API specializes its functionality depending on whether the node is a host or a switch, as explained in Chapter 2. For our *mcast* the only assumption made about the NodeServer functionality is its capability to establish and remove one unicast connection or one branch in a multicast connection, at a time. On request the MCM will access the *mcast* Router (MR) API. During mcast design nothing was assumed about the design of MR. It perfectly works together with the routing model explained in Chapter 3.

The initial design of *mcast* needs to be extended in order to make the service *manageable*. The management activity here relates to the OSI functional areas of performance and configuration management, and to the functions in the service management layer of the TMN architecture. The management task includes monitoring and controlling the service to ensure that it operates as intended, maintaining the service-level agreements while achieving management objectives. The task is performed on a slower time-scale than the functions executed in the service delivery system.

1.2.4 Transport Architecture

As part of this project a new object-oriented transport architecture has been designed to address the need Multimedia Networks have of a varied, flexible and QoS-aware set of transport protocols. In our architecture, transport components are dynamically bound at run-time to create a suitable transport protocol stack according to application QOS requirements.

The architecture consists of consumer/producer components that are separately represented by their transport abstraction--also called an *engine*, their control and management abstractions, and, a set of controllers implementing transport services such as QOS mapping and dynamic binding of suitable protocol stacks.

The consumer/producer components APIs defined in the current transport architecture are: the Network Provisioning (NetP), the Transport Protocol (TP) and Trabsport Multiplexer (TMux), believed to belong to the R/G::T.

The set of transport services APIs defined in the current architecture are the Protocol Stack Builder (PSB), the QOSMapper (QM) and the Transport Monitor (TM). PBS and QM are in the G::C and TM is in the G::N.The PSB is responsible for the construction of the protocol stack by binding a variety of engines together. It is aware of the variety of transport components supported on the end-system and the order in which they have to be bound together to create useful protocol stacks.

1.2.5 The Virtual Workshop Service

The Virtual Workshop is a distributed telelecturing application that allows multiple remote participants to collaborate in the style of an open workshop. Two modes of operation are supported in the application. In the first mode, there is a single sender and one or more receivers. The structure of the session in this case, takes the form of a multicast tree. In the second mode, there is only a single sender and receiver. The latter mode is useful in situations where only a point-to-point connection is needed. In the context of the workshop, the point-to-point mode is employed to implement a simple Video-On-Demand (VOD) feature where individual users receive a dedicated stream from a VOD server. In both modes, only uni-directional streams are supported.

The architecture of the virtual workshop consists of 3 logically distinct classes of the following components: the Virtual Workshop Graphical User Interface (VWGUI), the Telelecture Builder (TLB) and the Session Directory (SD).

1.2.6 Service Management Strategy

Service Management addresses the problem of designing and realizing manageable multimedia services. Our approach centers around defining a set of cooperating objects involved in the interaction between the service delivery and the service management system. The model we propose describes the interface between the two systems, and covers--in a generic way--the aspects of instantiation of a *service session*, its access by a user and its management by the operator. (A service session represents the information handled by all processes involved in providing a service; a session has a start, a duration and an end.) Our model suggests a generic solution to a recurring problem in service design. The model needs to be customized for a particular service and refined according to management requirements and available resources. In this sense, we are proposing a *design pattern* [GAM95] for making telecom services manageable.

This work focuses on service management in the sense of supervising the service delivery system. The management task includes monitoring and controlling the service delivery system to ensure that it operates as intended, maintaining the service-level agreements while achieving management objectives. The task is performed on a slower time-scale than the functions executing in the service delivery system. This activity relates to the OSI functional areas of performance and configuration management, and to the functions in the service management layer of the TMN architecture.

In the domain of service management we are considering, two activities can be identified: a) *managing the set of controllers* (i.e., routers, connection managers, etc.) which comprise the functionality of the service delivery system, and b) *managing the set of service instances* (such as audio multicast connections or VOD sessions) which are dynamically created, modified and terminated during the operation of the service delivery system. While we have presented some results on activity a) in [CHA96b], this paper focuses primarily on activity b), namely, on the management of service instances (or sessions) and aggregations of such instances.

We have applied our model for service management on a multicast VC service for a multiclass broadband network. To validate our approach and to gain experience, we have implemented the service and the management capabilities on two platforms, namely, a) on a high-performance emulation platform, which makes possible to develop a software prototype and to study its dynamic behavior and scaling properties in various scenarios [CHA96a], and b) a broadband testbed running xbind [LAZ96], a CORBA-based [OMG96] multimedia networking platform, on which services are fully implemented and the transport between multimedia devices is realized. In b) we use CORBA technology for building not only the service control system but also the service management system. CORBA provides a flexible object oriented environment for implementing distributed software systems, and it facilitates the integration of control and management software under a single technology.

1.2.7 Implementation Strategy: Corba and Telesoft

The initial implementation of the architecture described above has been done in a Corba-based environment and accessing the real resources in our ATM LAN in the lab.

In order to have access to our simulation environment a new software development environment was designed as part of this work: Telesoft. The connection management and service management building blocks have been implemented using Telesoft.

1.3 Rest of the Report

This report is organized as follows: Chapter 2 describes the APIs for using network switches and adapters. This activity corresponds to network resource control. Chapter 3 describes our model for routing. Chapter 4 describes how a connection management algorithm makes use of the APIs described in Chapters 2 and 3 to offer its APIs to a higher level service, such as a Virtual Workshop. But a higher level service needs more than connection management to be provided. Chapter 5 presents our transport model in detail and defines its APIs. Both connection management and transport APIs will be used to define higher level services. Chapter 6 describes an initial model for our Virtual Workshop service. Chapter 7 defines our model for service management focusing on the management of the connection management service. All of the above has been implemented in a Corba-based environment accessing real resources (an ATM LAN). As explained above, a new software development environment has been designed as part of this work; Chapter 8 describes the scope, design and implementation of Telesoft. The Appendix contains the current BIB.

2 APIs for Network Resource Control

The Network is a connected graph of switches and links. End-system stations connect to the network through a network adapter. The network architecture defines a set of concepts such as node, link, endpoint, traffic class and QoS. These definitions are presented in the Appendix as part of the BIB. Here we describe the APIs offered by the main components in our Network Architecture: the Adapter Server and the Switch Server.

2.1 APIs for Network Resource Control

Resource control is accomplished via the Adapter Server API, in the end-system, and the Switch Server API, in the network. Both the adapter and the switch are objects modeled as containers. The container object, with a well defined IDL, is inherited by the adapter and switch objects.

2.1.1 The Container

A container is an object that contains other objects. Containers can offer an additional set of IDL methods that group together several requests addressed to its components. This allows to reduce the CORBA signalling overhead: instead of binding to n different objects and addressing a request to each of them, the remote object binds only once to the container and transmits a single request to it. The request dispatching and method invocation on each component is then done locally by the container. Another way of improving efficiency is packing several similar requests in a single method invocation, by using sequences of parameters. For example, rather than asking a *SwitchFabric* interface to setup 100 channels one by one, a connection manager can invoke a single method that takes sequences as parameters on the switch server container.

The container can include both BIB interfaces and object references. The object references are visible only inside the container. The BIB objects and the container implementation objects may have a set of C++ methods that, even though public, are invisible from outside. In general, these C++ methods are used by the container components to communicate among themselves or with the container object. So, they provide an additional level of data encapsulation.

A container allows to group together and reuse BIB interfaces in different contexts. For example, the BIB interfaces described in the '`Core`', '`Capacity`' and '`Media Transporter`' sections can be grouped together with some non-CORBA object references inside two servers: an *Adapter Server* that models a workstation/PC network interface card and a *Switch Server*, that models the remote controller of a switch.

```
typedef sequence<VirtualResource> VirtualResourceSeq;
struct BIBObjRefInfo {
                        string bib_obj_ref;
                        string bib_interface;
                        string bib_obj_marker;
                        long creation_time;
};

interface VirtualContainer
{
  readonly attribute long no_interfaces;
  BIBObjRefSeq listAllBIBInterfaces();
  VirtualResourceSeq listByInterface(in BIBInterface bib_interface_name)
                        raises (Reject);
  VirtualResource getByMarker(in BIBObjMarker bib_marker_name,
                            in BIBInterface bib_interface_name)
                        raises (Reject);
  BIBObjRefInfo getInterfaceInfo(in BIBObjRef bib_obj_ref_name)
                        raises (Reject);
};
```

Types

The `BIBObjRefInfo` structure gives information about each BIB interface registered in the container:

```
struct BIBObjRefInfo {
  string bib_obj_ref;
  string bib_interface;
  string bib_obj_marker;
  long creation_time;
};
```

- `bib_obj_ref` - stringified pointer to the BIB IDL interface implementation.
- `bib_interface` - the name of the IDL interface.
- `bib_obj_marker` - marker of the interface implementation object.
- `creation_time` - time when the interface was registered.

Exceptions

- `BIBEx_NotFound` - raised if the container does not contain the interface or list of interfaces requested in one of the retrieval methods.

Methods

listAllBIBInterfaces. Returns a list of stringified pointers to the BIB IDL interfaces inside the container. If the container is empty, the list length is zero.

listByInterface. Returns a list of stringified pointers to the container BIB interfaces of a certain type. The type is given by the `bib_interface_name` string parameter. If there are no objects of the specified type, the list is empty (its length is zero).

getByMarker. Returns a reference to a CORBA object inside the container. The object is identified by its marker and by the type of interface it represents. The `bib_interface_name` is required to distinguish between two objects that implement different interfaces types, but have the same marker. If no objects matching the description are found, returns `NULL`.

getInterfaceInfo. Returns information (in the form of the `BIBObjRefInfo` structure) about an object reference input parameter. If the object reference is not part of the container, returns `NULL`.

2.1.2 The Adapter Server (AS)

An `Adapter` interface abstracts a network interface card. The API offered allows to establish and remove connections in it. Upon completion of a method, the adapter returns to the client the list of resources for which the operation was committed successfully. For reservations, the client can specify either that it wants all or nothing, or that a partial fulfillment of the request is acceptable. This is the role of the `atomic` parameter.

To understand what happens after a request arrives at the adapter, we need to look at the objects that are contained by the adapter; these will depend on the kind of network interface card we are talking about. An adapter connected to an ATM network contains three objects: two `ATM-NameSpace` interfaces (an input and an output name space) and a `SchedulableRegion` interface. The ATM name space interfaces monitor the available range of VP/VCs on the adapter, while the SchedulableRegion monitors the amount of bandwidth available. The detailed APIs offered by these objects are included in the Appendix.

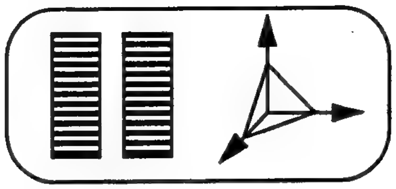


Figure 9 Internal structure of an Adapter Server

```
interface Adapter : VirtualContainer
{
  void reserveInputLinks(inout NameIdentifierList in_entries,
                         inout StringSeq auth_info,
                         in boolean atomic)
       raises (Reject);

  void freeInputLinks(in NameIdentifierList in_entries,
                         in StringSeq auth_info)
       raises (Reject);

  void reserveOutputLinks(inout NameIdentifierList out_entries,
                         in AnySeq connection_info,
                         inout StringSeq auth_info,
                         in boolean atomic)
       raises(Reject);

  void freeOutputLinks(in NameIdentifierList out_entries,
                         in AnySeq connection_info,
                         in StringSeq auth_info)
       raises(Reject);
};
```

Methods

reserveInputLinks . This is an example of method that multiplexes several requests into a single request. The parameter in_entries contains a list of entries of the input name space of the network interface card that an external client wants to reserve. Note that if only one entry needs to be reserved, or the entries form a compact range or match the same pattern, then this request can be addressed directly to the input name space interface. After the method has been executed, in_entries will contain the list of the name resources that have been acquired successfully.

freeInputLinks. The parameter `in_entries` contains a list of entries of the input name space of the network interface card that an external client wants to release. After the method has been executed, `in_entries` will contain the list of the name resources that have been freed successfully.

reserveOutputLinks. It reserves a set of naming and bandwidth resources on the output of a network interface card. The *n*-th element of the `out_entries` sequence represents the name identifier that the client wishes to acquire from the output name space. In our implementation, the *n*-th element of the `connection_info` sequence represents the amount of bandwidth the client requests from the schedulable region for the *n*-th connection. There are two possibilities: either the caller requests specific name identifiers, or it requests some name identifiers. In the second case, the adapter server will return to the caller the name identifier it has assigned to it in the parameter `out_entry`. The first request will fail if the specific name is already assigned to another client, while the second request will fail only if there are no more names available in the name space. Finally, when the method is executed the sequence out_entries will contain the names that have been successfully assigned, while the sequence connection_info will contain information about the resources assigned together with those names. If the `atomic` parameter is set to `TRUE`, and not all the requests could be satisfied, the two sequences have the length 0.

freeOutputLinks. It frees the resources acquired on an output link.

2.1.3 The Switch Server (SS)

A Switch interface models a remote switch/router controller. We have implemented the remote controller of an ATM switch. If the switch has *n* ports, then a switch server will contain the following BIB interfaces: *n* `Multiplexers`, *n* `SchedulableRegions`, *n* input and *n* output `VirtualNameSpaces` (one of each of these BIB interfaces per port), a `SwitchFabric` and a `MappingCapacity`. It will also contain the following object references: 2**n* (*n* input and *n* output) `ATMNameSpace` objects and a `qGSMPInterface`. The functionality of the BIB interfaces is detailed in the Appendix. The same object (`VirtualNameSpace` and `ATM-NameSpace`) is registered twice, as the objects inside the Switch Server have a different view on these objects from the objects outside. Finally, the `qGSMPInterface` object is being used to build, transmit, receive and decode qGSMP messages to or from the switch hardware.

In order to reduce the CORBA signalling overhead, a switch container can add/remove several connections or free/reserve several resources inside a single CORBA request. The method parameters are lists of names. The *n*-th elements of each list parameter represent the parameters

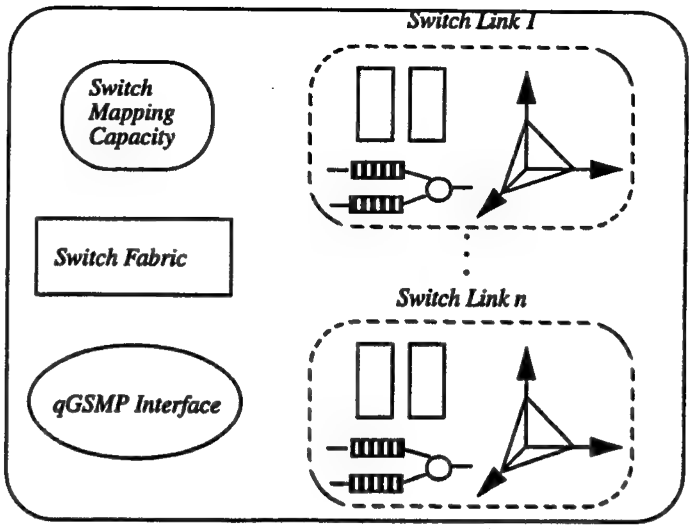


Figure 10 Internal structure of a Switch Server

for the *n*-th operation. This convention implies that, all the sequences that are passed as parameters to a method have to have the same length.

After the execution of a method, the switch server will return to the client the list of resources for which the operation was successful. This is why most parameters in these methods are declared as inout. The parameter atomic allows the client to specify that it wants all the requested resources, or nothing, if the answer to such a requests is negative, a set of empty lists is returned.

```
interface Switch : VirtualContainer
{
  void reserveOutputLinks(inout ShortSeq out_ports,
                          inout NameIdentifierList out_entries,
                          inout AnySeq connection_info,
                          inout StringSeq auth_info,
                          in boolean atomic)
       raises(Reject);

  void commitConnections (inout ShortSeq in_ports,
                          inout NameIdentifierList in_entries,
                          inout ShortSeq out_ports,
                          inout NameIdentifierList out_entries,
                          inout AnySeq connection_info,
                          inout StringSeq auth_info,
                          in boolean atomic)
       raises(Reject);

  void connect(inout ShortSeq in_ports,
                          inout NameIdentifierList in_entries,
                             inout ShortSeq out_ports,
                             inout NameIdentifierList out_entries,
                             inout AnySeq connection_info,
                             inout StringSeq auth_info,
                             in boolean atomic)
         raises (Reject);

    void freeOutputLinks(inout ShortSeq out_ports,
                             inout NameIdentifierList out_entries,
                             inout AnySeq connection_info,
                             in StringSeq auth_info)
         raises(Reject);

    void removeConnections(inout ShortSeq in_ports,
                             inout NameIdentifierList in_entries,
                             inout AnySeq connection_info,
                             in StringSeq auth_info)
         raises(Reject);

    void removeMulticastBranches(inout ShortSeq in_ports,
                                 inout NameIdentifierList in_entries,
                                 inout ShortSeq out_ports,
                                 inout NameIdentifierList out_entries,
                                 in AnySeq connection_info,
                                 in StringSeq auth_info)
         raises(Reject);
  };
```

Methods

reserveOutputLinks. It reserves resources on several output links in order to establish connections later. The `out_ports` parameter contains the list of ports on which the client wishes to reserve resources. The `out_entries` parameter contains the list of name identifiers to be acquired. The `connection_info` parameter contains the list of the amounts of bandwidth requested by each reservation from the respective schedulable region.

commitConnections. This method is called after to establish connections after `reserveOutputLink` was performed successfully. For each connection, it acquires the name identifiers on the input ATM name space, checks with the `MappingCapacity` interface if the routing table has free entries; if there is space left on the routing table, the method sends a request to the `SwitchFabric` interface to add a new channel. The switch fabric will ask the `qGSMPInterface` object to build a qGSMP message (`addBranch`) and will send it to the physical switch. If the result of all these operations is successful, a new connection is established on the switch.

connect. This method has the functionality of both previous methods.

freeOutputLinks. It frees the resources acquired on a set of output links before removing a set of connections. For each connection, it frees the ATM name identifier of the output ATM name space and the bandwidth that was acquired from the schedulable region.

removeConnections; For each entry in the lists, releases the name identifier of the input ATM name space and removes the channels established on the switch for the connection whose input entry was released. Updates the status of the switch mapping capacity. This method removes a either a multicast tree, or a set of unicast connections. It assumes that the caller knows how many branches are rooted in that connection, and what are the parameters of each branch. In particular, the length of the `connection_info` and `auth_info` sequences has to be equal to the number of output branches corresponding to the trees rooted in each of the elements of the `in_ports` and `in_entries` sequences.

removeMulticastBranches. It removes a set of branches of one or several multicast connections. The channel established on the switch for each branch, as well as the resources on each output port, are released. If this is the last branch in a multicast tree, then the naming resources associated with the root of the tree are also released.

2.2 Implementation Aspects

Adapter Boot-up Procedure

The adapter server instantiation code calls the constructors of all its components. Each BIB interface that is being instantiated has to be registered with the container. If the implementation of the BIB interface contains C++ methods that are used inside the server, than the same object has to be registered twice, once as a BIB interface and once as an object reference. For example, the `ATM-NameSpace`, the implementation of the `VirtualNameSpace` IDL interface, needs to be registered as a `VirtualNameSpace` BIB interface, as well as an `ATMNameSpace` object reference, in order to use implementation specific methods inside the adapter container

Switch Boot-up procedure

In order to start the switch server, one needs to specify a *host name*, identifying a machine on which the CORBA switch server will run and a *control VCI* used to send qGSMP messages to the switch.

The `qGSMPInterface` object reference is built and registered with the container. It sends a `SwitchConfiguration` message to the switch. The response from the ATM switch will contain an unique identifier: its MAC address. The `qGSMPInterface` sends then an `AllPortsConfiguration` message to the switch. The reply to this message will return the number of ports of the switch, as well as information about each of these ports: VPI/VCI range supported, the interface type, the cell rate supported.

The `SwitchFabric` and the `MappingCapacity` are instantiated at this point. It is assumed that a unicast connection requires one slot in the switch fabric, while a multicast connection requires a number of slots equal to the number of ports of the switch.

Once the port information is received, the ATM name space objects can be built for each available port. Each ATM name space will be characterized by a VPI/VCI range supported on that port. If the port is uni-directional, then a single ATM name space needs to be instantiated. If it is bi-directional, then two ATM name spaces are instantiated for each port: one for input and one for output. The name spaces are registered twice with the container: once as `VIrtualNameSpace` BIB interfaces, and once as `ATMNameSpace` object references. The first interface is used by external clients (such as Connection Managers) in order to acquire different name identifiers. The second interface is used by objects inside the switch server, in order to keep information about the internal state of the switch.

Next, the `qGSMPInterface` sends a `QOSPortConfiguration` message to the first available port on the switch. This will allow to test if the switch supports the QOS extensions to the GSMP protocol. If the switch supports qGSMP, then, `PortQOSConfiguration` messages will be sent to each port of the switch. If it does not support qGSMP, then the output multiplexers (one per port) have to be initialized using a trivial configuration. A `SchedulableRegion` interface is instantiated for each port, using the cell rate information (returned by `AllPorts-Configuration`) in order to determine the total capacity of each port. If qGSMP is supported, then the Multiplexer and SchedulableRegion interfaces can be configured in order to support certain cell level QOS constraints.

3 APIs for Routing

Routing is an integral part of any telecommunication network. Originally intertwined with other communication services, routing has evolved into an independent entity providing services to other agents such as connection managers. During the course of this evolution, the principal requirements of dynamic routing have remained the same, namely state distribution and route computation. However, with the dramatic progress in computing systems technology, the level of sophistication realizable has improved dramatically. Thus, high level abstractions, both in terms of computing and communications, can be realized. By masking low-level details and providing an elegant programming interface, these abstractions help software developers visualize networking resources in a natural and intuitive form. Open interfaces between various functional modules permit each of them to be plugged in independently. With product differentiation in the telecommunications industry increasingly depending on flexibility and speed of deployment, such abstractions and open interfaces are vital for success.

Previous work on routing architectures with quality of service has focused on low-level protocols akin to assembly language. Examples include PNNI [PNN96], RSVP/QOSR [BRA97, CRA98, GUE98, ZHA97], and CCS #7 [RUS95]. Such protocols, though efficient like assembly language, are significantly more difficult to program and pose difficulties for future expansion and changes. In this chapter, we present a routing architecture and IDL API that provides high-level abstractions and a clear separation between state distribution, route computation, connection management, and switching hardware. Each functionality may thus be plugged in by different software providers. In the present market, APIs to router and switch hardware are closed, thus giving hardware vendors monopolistic control over software. However, they do provide peer-to-peer APIs at the level of connection management and state distribution, in contrast to the switch level API presented in Chapter 2 and qGSMP [ADA97]. This in effect determines the routing architecture and much of the associated algorithms and mechanisms. In contrast, the API presented in this chapter standardizes only the access to objects, not the algorithms that operate on these objects.

This gives the flexibility for the introduction of multiple algorithms and mechanisms to compete and coexist.

3.1 Model for Routing

Figure 11 shows a detailed view of the routing system based on objects and APIs. The four objects inside the dotted box form the routing service. The state distribution service is depicted on the left, and the connection management service is depicted on the right.

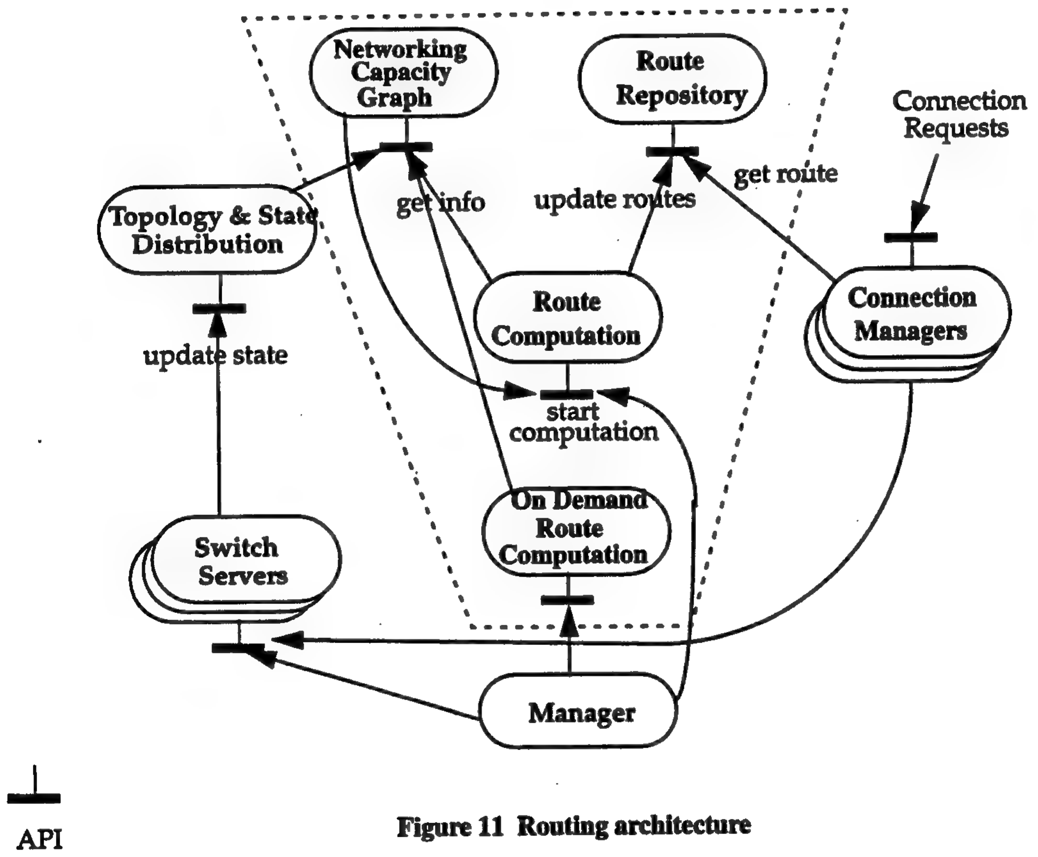


Figure 11 Routing architecture

The routing service is a replicated functionality with each of the four objects in Figure 11 appearing at multiple nodes in the network. Thus, it has no single point of failure, and in addition connection managers can access the routing service at multiple nodes. Thus, if the routing service at the closest node fails, connection managers can still access routes from a distant node. Route computations are carried out entirely locally, and in parallel at the various nodes. This is enabled by the state distribution service which collects network resource state by accessing the switch servers throughout the network and presents it to the routing service.

We now discuss the four objects that comprise the routing service. The Networking Capacity Graph (NCG) object is a database for the network topology, schedulable regions and other link-state information. The Route Repository (RR) is a database for storing routes pre-computed by an algorithm running within the Route Computation (RC) object. These routes are categorized according to a finite set of traffic classes. The distinction between NCG's and RR's lies in the fact that the former contain information on individual network resources while the latter contain information on groups of resources. For example all state information related to individual links is to be found in NCG's whereas information related to routes (sequences of links) such as the call-level traffic load and load balancing parameters is to be found in RR's. The "On-demand route computation (ODRC)" objects are used for computing routes with traffic and QOS descriptions that do not correspond to any traffic class used by the Route Repository. Additionally, slow timescale operations such as the computation of routes for creation of virtual paths (VPs) and virtual networks (VNs) is also handled by the ODRC. Figure 11 shows a manager setting up a VP: it first accesses ODRC and then the necessary switch servers.

Different mechanisms are possible for use in state distribution. Two main possibilities are flooding and the tree-based distribution. Another possibility which hides the actual distribution mechanism is to use the CORBA event service. The implementation of the event service will however rely on either flooding or tree-based forwarding. The event service is an attractive option since it hides the underlying mechanism, thus providing a simple interface. However, our aim is not to standardize the state distribution mechanism. Thus, we will not address this issue any further. Given the APIs for the routing service, any state-distribution mechanism can be implemented by a third party.

The operation of the routing system is as follows: Each switch server monitors the state of all the links attached to its switch. Periodically, the switch server sends state updates to the state distribution service which writes the updates into the various NCG's in the network. Whenever there is a significant state change, the switch server immediately sends an update to the state distribution service, so that the routing service can take immediate action. Such a distinction between significant state change and ordinary state change is important for quick reaction to changes in the network. At present, the only state change categorized as significant is a link failure. In the future, other changes such as a schedulable region shrinkage, would also be added to this category. When an NCG receives a state update, it checks whether the update is significant. If it is, the NCG invokes its associated RC immediately to recompute routes. Additionally, the NCG invokes the RC periodically provided there has been some state change. The RC's may use any algorithm for route computation. Moreover, RC's at different nodes may use different algorithms. The RC's write the routes they compute to their respective RR's, from which connection managers may read. The control cycle is completed when the connection managers access the switch servers along the route read from and an RR in order to setup a connection.

3.2 APIs for Routing

The data structures used by the routing APIs are in the Appendix as part of the BIB. They include the concepts of traffic descriptor, QoS, traffic class, node, link, link state, switch capacity, cross connect, schedulable region, and route as a sequence of cross connects.

`Routes` are represented as sequences of `CrossConnects`. The structure `RouteDistribution` places a probability distribution on a set of routes used for load balancing. A `CrossConnect` is defined as the connection from an input port to an output port of a switch. The element `CrossConnect::outport_cr_id` identifies contract region of the underlying link if the output link in concern is not a physical link.

All `Links` are unidirectional point-to-point. The element `Link::src_cr_id` identifies contract region of the underlying link if the link in concern is not a physical link. When links are grouped together into a `LinkSequence`, they should be "breadth-first-search" ordered. The `LinkState` refers to statistics computed over a period specified external to the structure. The structures `LinkTCDescription`, `LinkSRDescription`, `LinkStateDescription`, and `SwitchCapacityDescription` are used by the event service to distribute state information.

3.2.1 Networking Capacity Graph (NCG)

```
interface NetworkingCapacityGraph
{
  void      SetLocalAddress(in IPAddress node);
  IPAddress GetLocalAddress();
  void      AttachRC(in RouteComputation rc);


  void AddNode(in IPAddress node) raises(NodeAlreadyExists);
  void DeleteNode(in IPAddress node) raises(NonExistentNode);
  void DeleteAllNodes();
  IPAddressSequence GetAllNodes();
```

```
    void AddLink(in Link link)               raises(NonExistentNode);
    void AddLinks(in LinkSequence links)     raises(NonExistentNode);
    void DeleteLink(in Link link)            raises(NonExistentLink);
    void DeleteLinks(in LinkSequence links)  raises(NonExistentLink);
    void DeleteAllLinksFromNode(in IPAddress node)
         raises(NonExistentNode);
    void DeleteAllLinks();

    LinkSequence GetAllLinksFromNode(in IPAddress node)
                 raises(NonExistentNode);
    LinkSequence GetAllLinksToNode(in IPAddress node)
                 raises(NonExistentNode);
    LinkSequence GetAllTwoWayLinksFromNode(in IPAddress node)
                 raises(NonExistentNode);
    LinkSequence GetAllLinksBetweenNodes(in IPAddress src,
                 in IPAddress dest) raises(NonExistentNode);
    LinkSequence GetAllLinks();

    void      SetLinkState(in Link link, in LinkState state)
              raises(NonExistentLink, InvalidLinkState);
    LinkState GetLinkState(in Link links) raises(NonExistentLink);

    void SetSchedulableRegion(in Link link,in CapacityPlaneList sr)
         raises(NonExistentLink, InvalidSchedulableRegion);
    CapacityPlaneList GetSchedulableRegion(in Link link)
                      raises(NonExistentLink);

    void SetTrafficClasses(in Link link,
                           in TrafficClassSequence traffic_classes)
         raises (NonExistentLink, InvalidTrafficDescription,
                 InvalidQOS);
    TrafficClassSequence GetTrafficClasses(in Link link)
                         raises(NonExistentLink);
};
```

The methods `SetLocalAddress` sets the address of the machine on which the `Net-workingCapacitGraph` object is residing, and `GetLocalAddress` returns the previously set value. The method `AttachRC` specifies which `RouteComputation` object to call when the network state changes. The add/delete node methods are self-explanatory. In case an exception is raised while deleting a sequence of links, only a partial list of the links may get deleted. After determining the cause of the exception, the method should be called again to ensure that all the links are deleted. The method `DeleteAllLinksFromNode` deletes all links that originate from the specified node. Links that terminate at the specified node are not deleted.

The method `GetAllLinksFromNode` returns all links originating from the specified `node` and the method `GetAllLinksToNode` returns all links terminating at the `node`. The method `GetAllTwoWayLinksFromNode` returns all links originating from the specified `node` such that there is a link in the opposite direction. The method `GetAllLinksBetween-`

Nodes returns all links originating at src and terminating at dest. Finally, the method GetAllLinks returns all links in the database.

The methods SetLinkState, SetSchedulableRegion, SetTrafficClasses and their corresponding "get" methods are used to set/get the state, schedulable region and traffic classes on a given link.

3.2.2 Route Repository (RR)

```
interface RouteRepository
{
  void AddNode(in IPAddress node) raises(NodeAlreadyExists);
  void DeleteNode(in IPAddress node) raises(NonExistentNode);

  void SetTrafficClasses(in TrafficClassSequence classes)
       raises(InvalidTrafficDescription, InvalidQOS);
  TrafficClassSequence GetTrafficClasses();

  void     SetCallLoad(in IPAddress src, in IPAddress dest,
                       in CallLoad load)
           raises(InvalidCallLoad);
  CallLoad GetCallLoad(in IPAddress src, in IPAddress dest)
           raises(NonExistentNode);
  CallLoadSequence GetCallLoadFromNode(in IPAddress src
                                  out IPAddressSequence dest)
           raises(NonExistentNode);

  void SetRouteDistribution(in IPAddress src, in IPAddress dest,
                            in short traffic_class,
                            in RouteDistribution distribution)
    raises(NonExistentNode, InvalidTrafficClass,
           InvalidRouteDistribution);

  Route GetRoute(in IPAddress src, in IPAddress dest,
                                 in TrafficDescription traffic, in QOS qos)
        raises(NonExistentNode, InvalidTrafficDescription,
               InvalidQOS);
  Route GetRoute2(in IPAddress src, in IPAddress dest,
                  in short traffic_class)
        raises(NonExistentNode, InvalidTrafficClass);

  RouteDistribution GetRouteDistribution(in IPAddress src
                                    in IPAddress dest,
                                   in TrafficDescription traffic,
                                     in QOS qos)
        raises(NonExistentNode, InvalidTrafficDescription,
               InvalidQOS);
  RouteDistribution GetRouteDistribution2(in IPAddress src,
                                      in IPAddress dest,
                                    in short traffic_class)
       raises(NonExistentNode, InvalidTrafficClass);

  void DeleteAllRoutes();
};
```

The method `SetTrafficClasses` specifies the set of end-to-end traffic classes for which routes are to be computed. The method `SetCallLoad` sets the call load of the network. The corresponding "get" methods are used by routing algorithms to retrieve the traffic class descriptions and network loading. The method `GetCallLoadFromNode` returns the load corresponding to all destination nodes such that the source node is the specified `node`; the set of destination nodes is returned in `dest`.

The method `SetRouteDistribution` sets a route distribution (see section on data structures) corresponding to the given `src-dest` pair and `traffic_class` number. The method `GetRouteDistribution2` returns route distribution set by `SetRouteDistribution`. The method `GetRouteDistribution` returns the distribution corresponding to a traffic class which has the same traffic description as specified in the request and a QOS that is at least as good as that specified in the request. Subject to that restriction, the implementation is free to choose any traffic class. The two methods `GetRoute` and `GetRoute2` are similar to the above two methods. The difference is that they return a single route selected from the distribution. Successive calls may return different routes, based on the distribution probabilities. Finally, the method `DeleteAllRoutes` clears the route database.

3.2.3 Route Computation (RC)

```
interface RouteComputation
{
  void AttachNCG(in NetworkingCapacityGraph ncg);
  void AttachRR(in RouteRepository rr);
  RoutingAlgorithmTypeSequence GetSupportedRoutingAlgorithms();
  void SetRoutingAlgorithm(in RoutingAlgorithmType type,
                           in any parameters)
    raises(InvalidParameters);
  oneway void ComputeRoutes();
};
```

The methods `AttachNCG` and `AttachRR` are for specifying which NCG and RR to use for route computations. The method `GetSupportedRoutingAlgorithms` returns a list of routing algorithms that are supported by the implementation. The method `SetRoutingAlgorithm` sets the algorithm to be used. The method `ComputeRoutes` triggers a route recomputation.

3.3 Comparison with Existing Architectures

First and foremost, the API for the routing architecture presented above is based on CORBA IDL as opposed to the low-level protocol definitions provided by PNNI and other similar protocols. Thus, programming these APIs becomes much simpler and vastly more powerful. The ability to seemlessly access remote objects and the automatic marshaling of arguments provided by CORBA relieves the programmer of tedious chores, thus letting him focus on the actual application.

The high-level QOS abstraction provided by the schedulable region gives the programmer an intuitive capacity concept similar to that in multi-rate circuit switching [HYM91, LAZ98, ROS95]. Thus, call admission control becomes a trivial task. Moreover, much of the know-how in routing in circuit switched networks carries over to packet switched networks characterized by schedulable regions. In essence, the schedulable region abstracts cell-level phenomena and presents a simple interface for use by call-level control algorithms. On the contrary, the low-level QOS characterization found in PNNI and RSVP do not provide such an interface. By making the schedulable region an integral part of the routing architecture presented in this chapter, we make the task of the programmer simpler and more intuitive.

The API for the routing system described above was designed with well-defined interfaces so that third-parties could write and plug in their own routing algorithm or state distribution service without being concerned with other aspects of the system. In particular, the interfaces permit each of these algorithms to run on even different machines. In contrast, PNNI does not provide any interfaces between the state distribution, route computation, and connection management. Thus, all three functionalities within a given switch must be implemented together. The same constraint applies to the RSVP/QOSR protocols suites. Though these protocols provide well-defined interfaces for peer-to-peer communication across routers/switches, they provide nothing for communication between different functional entities (except through a limited management interface) (Figure 12). As a result these functional entities must be bundled together.

Moreover, since PNNI and RSVP provide peer-to-peer APIs at the level of connection management and state distribution, mechanisms and state machines must be standardized. In contrast, the APIs presented here impose no such restriction, thus permitting for example both parallel and serial connection management mechanisms to coexist. The resulting open architecture is vastly more flexible.

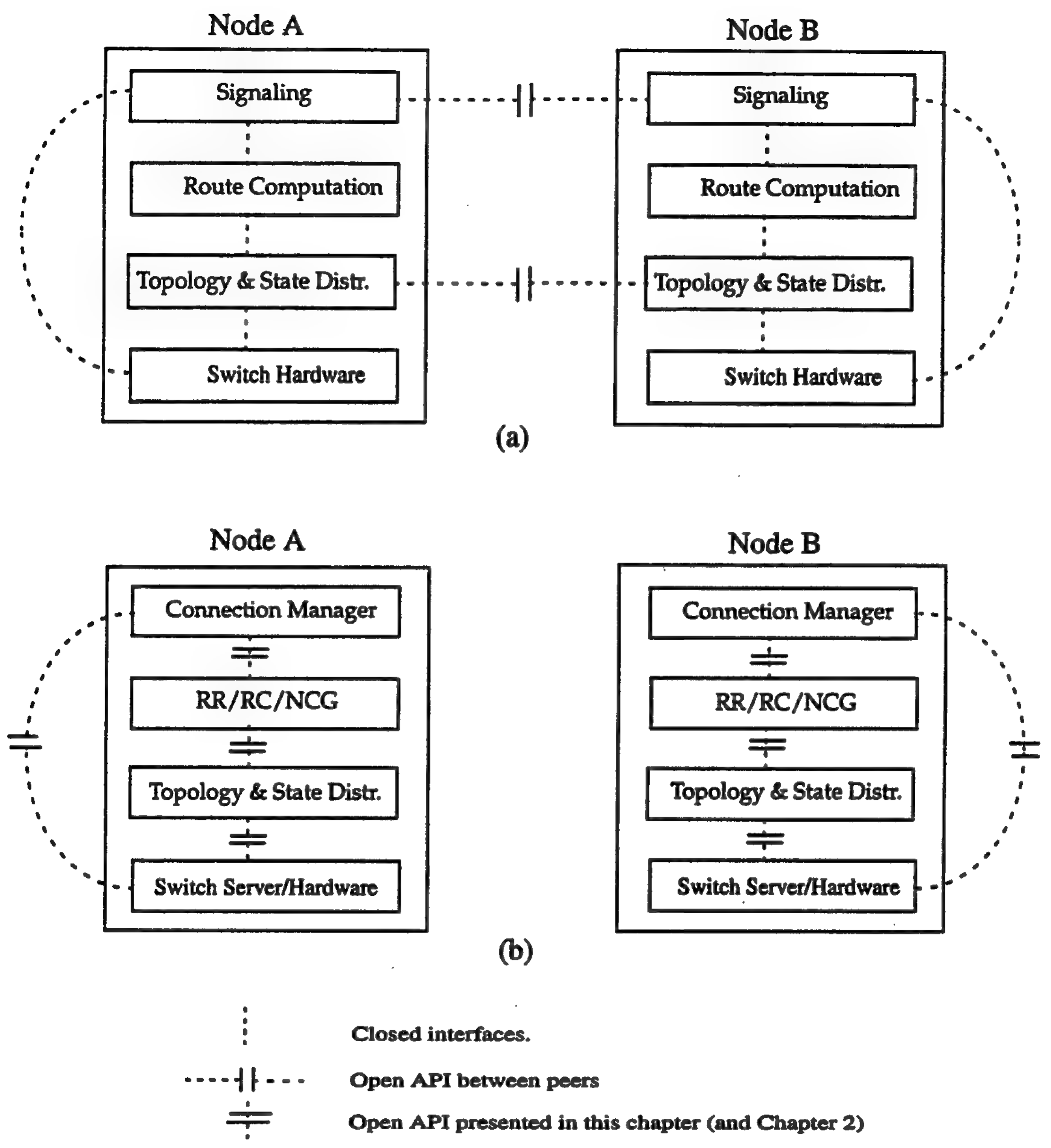


Figure 12 The peer-to-peer APIs provided by PNNI and similar protocols is shown in (a). They do not provide open APIs between different functional entities. Figure (b) shows the APIs presented in this chapter, which provide interfaces between different functional entities but not between peers. The four boxes in each of the nodes shown in (b) correspond to the same four in (a). For convenience of comparison, some of the boxes in (b) have multiple objects.

4 APIs for Connection Management

Connection Management is in charge of binding network resources to provide end to end connections over a network. An object called connection manager (CM) will receive the request to initiate the binding. The binding will be accomplished via requests to individual network switches and adapters, via the corresponding switch servers and adapter servers, once the route has been decided. As part of this project a specific connection management algorithm has been designed: *mcast*.

mcast is an ATM multicast service based on an object oriented design. It provides the capability of transporting multimedia information from a source to one--or more than one--destination. An *mcast* session belongs to one specific service class, i.e., video, data or voice. (A service class defines the performance characteristics and the performance requirements of a cell flow.) An *mcast* session can be represented as a tree where the root is the source of information and the leaves are the sinks (aka. receivers, destinations)[1]. A destination can request to join or leave an *mcast* session. The termination of a session is requested explicitly by the source.

Given an *mcast* session, each source and destination is identified by a host and a stream interface. The stream interface represents the "port" through which the information flow is sent or received. An *mcast* session is uniquely identified by its source (host Id and stream interface Id), which becomes its session Id. This session Id is of global scope.

In ATM the information being transported is a cell flow and the stream interface is defined by an ATM triplet (port-vpi-vci). An *mcast* session belongs to a specific service class. The service class specifies the traffic description and the expected end to end QoS.

1. A session can exist without destinations.

Four operations are offered to an *mcast* service user:

- initiate(**in** serviceClass, **out** sessionId, **out** streamInterface)
- join (**in** serviceClass, **in** sessionId, **out** streamInterface)
- leave(**in** sessionId)
- terminate(**in** sessionId)

In the design of *mcast* we have worked under the assumption that control messages are delivered over a reliable FIFO channel. This allows us to isolate concerns and focus primarily on strict service functionality. Still issues of concurrency under asynchronous conditions, as is the case for *mcast*, lead to a system behavior difficult to understand making the proof of its correctness a difficult task. This is addressed in Section 2.3.

Our purpose in designing *mcast,* an ATM connectivity service, is to experiment with our ideas on service management. Based on our current service management model [AUR97a], a management system for *mcast* has been designed and implemented. We design and implement the service and its management on two different platforms: a simulation platform based on parallel DES (discrete event simulation) [CHA96a] and a CORBA-based platform (*xbind*) [LAZ96]. Our software development environment is TeleSoft, which allows us to write the code independently of the platform on which we run it. Telesoft is currently being implemented as a prototype.

The *mcast* service capabilities are similar to those defined by the ATM Forum [ATMF95] or the IETF [ZHA93]. For instance, *mcast* allows for root, leaf and third-party initiated join and leave requests; contrary to the IETF scheme, an *mcast* session stays alive until it is explicitly terminated.

4.1 *mcast* design

We make use of a high-level pseudo-IDL to describe the APIs.

4.1.1 Objects, interactions and interfaces

mcast design includes three classes of controller objects: the *mcast* directory service (MDS), the *mcast* router (MR) and the *mcast* connection manager (MCM). No controller has a global view of the ongoing sessions; knowledge is truly distributed. Figure 13 shows the number of instances of each controller necessary to provide the service.

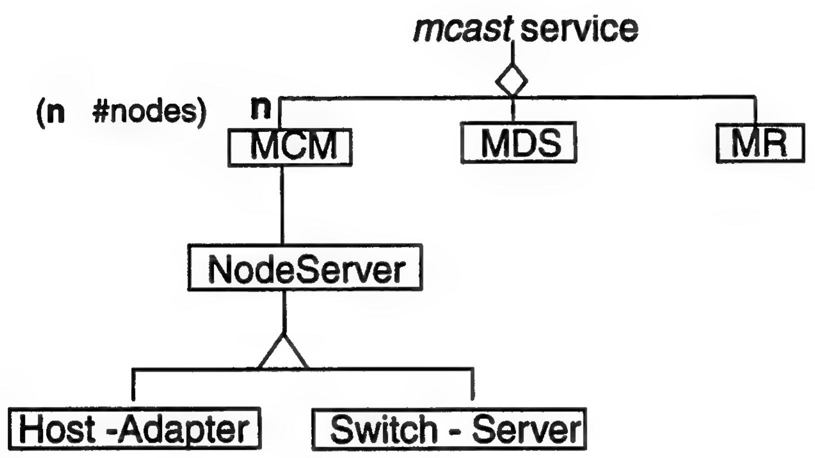


Figure 13 Cardinality relationship among *mcast* controllers.

- MDS is providing two functions: (1) allow prospective destinations to get information about the existing *mcast* sessions; (2) keep partial information about each existing *mcast* tree, which will be used for routing purposes when new destinations join.
- MR computes the route to a prospective destination upon request. Out of all the nodes in the tree, it selects the one which will act as source in the new branch.
- There is one MCM instance associated to each node (host or switch). The operations offered by MCM are associated with two interfaces. One is the service user interface mentioned in the previous section that will be offered by MCMs at the periphery (hosts). The other is a peer interface that allows MCMs to interact with each other (at hosts and switches).

The NodeServer is not, strictly speaking, part of *mcast*. It realizes the interface between the *mcast* service delivery system and the network resources. It will later specialize its functionality depending on whether the node is a host or a switch, as shown in Figure and as explained in Chapter 2. The only assumption made about the NodeServer functionality is its capability to establish and remove one unicast connection or one branch in a multicast connection, at a time.

Figure 14 shows how objects interact via control interfaces. All the interactions are asynchronous. Given their distributed nature, we believe telecom services need to be designed as a set of asynchronously interactive controllers. This design decision will also help in re-using the design in our two platforms, since objects naturally interact in an asynchronous manner in the emulation environment. Because routing is not a concern in our current design work, the router

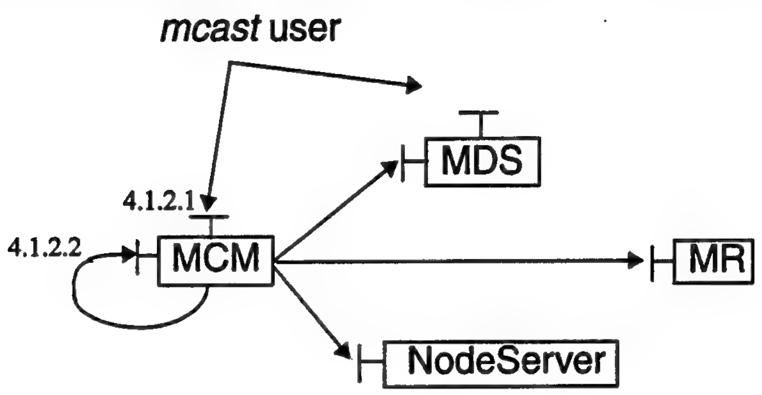


Figure 14 Invocation relationship between *mcast* object classes.

(MR) is not keeping any utilization state. Because of its simplicity, our current implementation has one router per MCM; MR and MCM interact synchronously via a function call:

Because routing is not a concern in our current design work, the router (MR) is not keeping any utilization state. Because of its simplicity, our current implementation has one router per MCM; MR and MCM interact synchronously via a function call:

- getRoute(**in** possible sources, **in** destination, **out** route).

No assumptions are made about the routing algorithm.

The NodeServer (NS) encapsulates the switch/adapter capabilities to establish and terminate local connections. The assumed interface for NS is as follows:

- connect(**in** serviceClass, **in** inEp, **in** outPort,**out** outEp)
- disconnect (**in** inEp, **in** outEp)

Both requests apply to either a branch or a unicast connection.

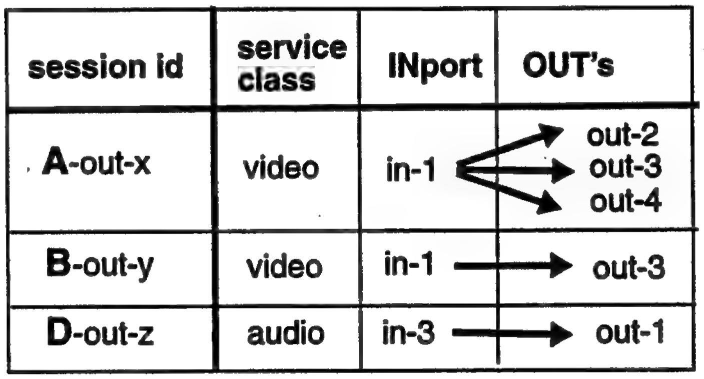
Operation signatures are shown using a CORBA-like high-level description language, because it helps the reader to understand at a glance what the message does. Each signature gives the operation name and the list of **`in`** and **`out`** parameters. Most operation names reflect the action requested on the server[1]; `leavingNode()` -shown in 2.1.1.2 is an exception because its name refers to the event happening at the client side: a node is leaving the tree; the action the server takes as a result depends on the server state.

Next we describe MCM and MDS in more detail.

4.1.1.1 MCM (*mcast* Connection Manager)

An MCM at a node (host or switch) keeps a table with information of the ongoing *mcast* sessions of which the node is a part. The example in Figure 15 shows the table reflecting a switch taking part on three *mcast* sessions originated at hosts **A**, **B** and **D**. For the session originated at host **A**, the cell flow is being multicasted from port 1 to ports 2, 3 and 4.

The MCM interface can be split into two parts (as in Figure 14): one offered to an *mcast* user and the other offered to a peer MCM.



session id	service class	INport	OUT's
A-out-x	video	in-1	out-2 out-3 out-4
B-out-y	video	in-1	out-3
D-out-z	audio	in-3	out-1

Figure 15

MCM interface offered to the *mcast* service user (mentioned in Section 1):

- **initiate (in serviceClass, out sessionId, out streamInterface)**

1. In this context server refers to an object receiving a message sent by another object.

- join (**in** serviceClass, **in** sessionId, **out** streamInterface)
- leave (**in** sessionId)
- terminate (**in** sessionId)

`initiate` and `terminate` are offered by the MCM at source, while `join` and `leave` are offered by the MCM at destination. `initiate` and `join` return a streamInterface (ATM endpoint) to/from which the *mcast* user (source/destination) sends/receives the information flow.

At `initiate` request, the MCM at source makes the necessary reservations in its domain (host), and registers the new session at MDS with the selected sessionId. The sessionId is composed of the source's host id and the ATM port-vpi-vci allocated for sending the flow. Deciding when the transport of information actually starts is out of the scope of *mcast*.

At `terminate` request the MCM at source frees resources in its domain, updates the information in MDS and sends the message (`terminate`) down the tree.

At `join` request the MCM at destination initiates a connection (branch) setup procedure.

At `leave` request the MCM at destination initiates a hop-by-hop branch termination.

See 2.2 for details.

MCM interface offered to a peer MCM:

- replicateFlow (**in** sessionId, **in** outPort, **out** outEp)
- joinNode (**in** sessionId, **in** serviceClass, **in** inEp, **in** outPort, **out** outEp)
- leavingNode (**in** sessionId, **in** myNodeId)
- `terminate (in sessionId)` – same as above

`replicateFlow` and `joinNode` are used at connection (branch) setup. `replicate-Flow` is sent to the *joining node*, the node in the existing tree to which the new branch is attached (it replicates the flow). `joinNode` is sent to the rest of the nodes in the route assigned to the new branch. `joinNode` specifies a serviceClass parameter that has a local meaning. Obviously there is need for some mapping functionality between e2e and local `serviceClass`, although this is not addressed in this report.

`leavingNode` is the message passed up the tree when a destination leaves the tree. The meaning of this message is: "stop sending me the flow". The parameter `myNodeId` identifies the sender of the message; it is needed by the receiver (object acting as server) to identify which branch must be removed.

If a `replicateFlow` message arrives at a node which is already replicating the flow to the output port requested, the MCM replies with the vci number being used; if a `replicateFlow` message arrives at a node which does not take part of the session by the time the `replicateFlow` message arrives, the request is rejected.

If a `leavingNode` or a `terminate` message arrives at a node that does not belong to the tree, no further action will take place. These scenarios are possible because of the concurrency of users. Concurrency issues are discussed in Section 2.3.

4.1.1.2 MDS (*mcast* Directory Service)

MDS is providing two functions: (1) allow prospective destinations to get information about the existing *mcast* service sessions; (2) keep partial information about each existing *mcast* tree, which will be used for routing purposes when new destinations join. Having these two kinds of information in the same object simplifies the task of keeping the consistency among them. Scalability issues need to be studied further.

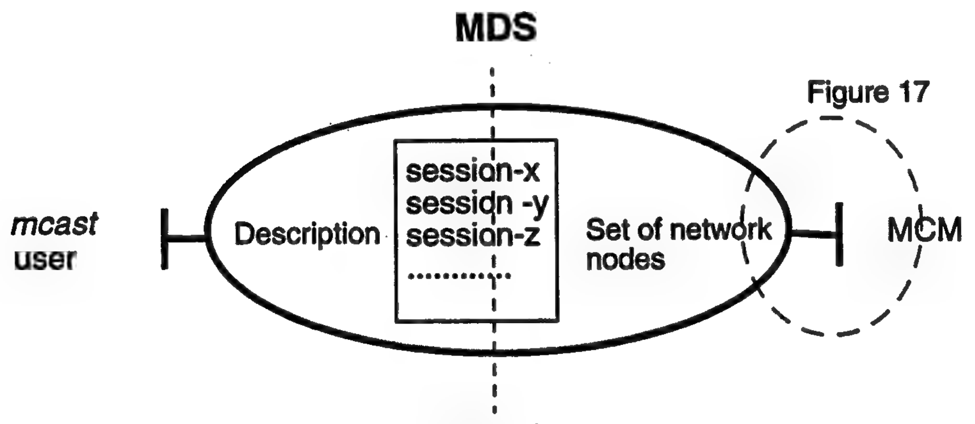


Figure 16

The *mcast* user can get the list of existing sessions.

- `getCurrentSessions (out listOfSessions);`

This functionality might belong to a higher-level service which uses *mcast* (see Section 5). This means that the *mcast* user might have obtained this information elsewhere or just be told by some other entity what message to send to the MCM.

The set of interactions between MDS and the MCMs is summarized in Figure 17.

u(x) refers to an update operation for x. = i, j, l, t[1], while q() corresponds to a query operation.

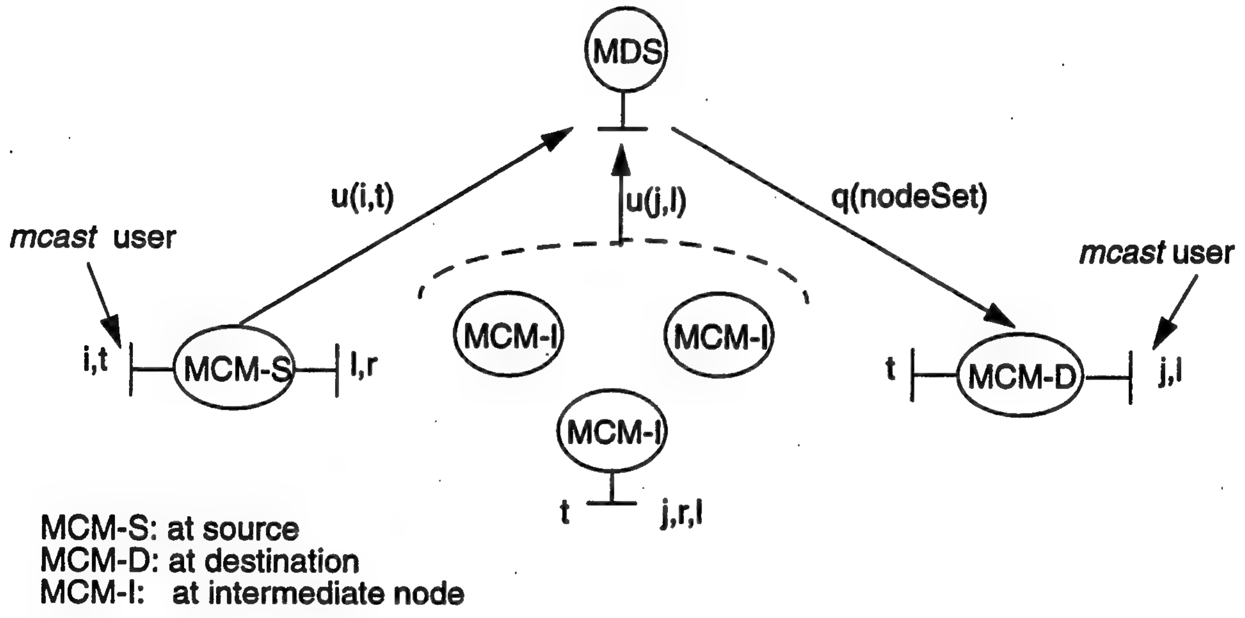


Figure 17

MDS interface offered to MCM at source:

- register (in sessionId, **in** description);
- deregister (in sessionId)

1. initiate, join/joinNode, leave/leavingNode and terminate

`register` and `deregister` are used by MCM at source as a result of an `initiate` or `terminate` operation to update the set of sessions kept at MDS. The `description` parameter should include at least the `serviceClass`.

MDS interface offered to MCM at destination:

- getNodeSet (**in** sessionId, **out** nodeSet);

`getNodeSet` is used by MCM at destination when a `join` request arrives.

For each *mcast* session MDS keeps the set of nodes involved in the tree: *nodeSet*. This information is used by the router; it provides the router with a set of possible joining nodes in the existing tree when a new destination wants to "join" the tree. Nodes in the set must therefore be able to relay and multicast the cellflow from an input port to one or several output ports when necessary. This is why only switches, and not hosts, are in the nodeSet.

MDS interface offered to MCM at switch:

- addToNodeSet (**in** sessionId, **in** nodeId);
- removeFromNodeSet (**in** sessionId, **in** nodeId);

`addToNodeSet` and `removeFromNodeSet` are messages sent by MCMs at switches when they join and leave the tree, respectively. For this update system to work correctly we need to assume a reliable FIFO channel.

4.1.2 Join, leave and terminate scenarios

For the next examples we consider the setup in Figure 18.

MCM1 is associated to host 1, that wants to become a destination in an existing *mcast* session. MCM2 is associated to switch 2, a switch that is already a member of that *mcast* session. MCM3 is associated to switch 3 that lies on the path from switch 2 to host 1.

(Arrows in Figures 18 and 21 represent transport of information, as opposed to messages between *mcast* controllers.)

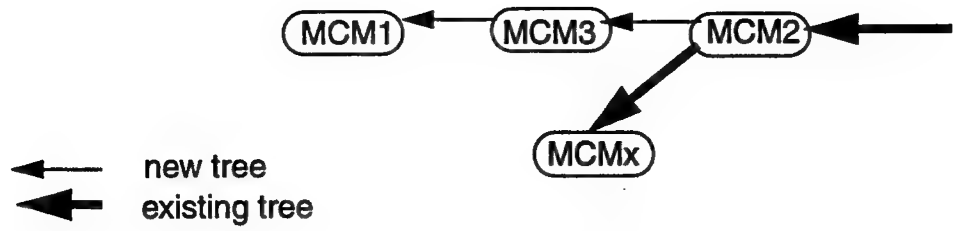


Figure 18

Because, in fact, the interactions are asynchronous, operations described up till now with `out` parameter(s) represent in reality two messages/methods, one for the request and the other for the reply. The method names have -Req and -Rep at the end to reflect this. There is one case - `leavingNode`- which is not considered a request but an indication[1], because it merely informs of the occurrence of an event (a node leaving the tree); it does not explicitly request an action. In this case the method name is ended with -Ind.

Indications have no reply by definition. Requests might have or not a reply. A request will be matched to its reply -if any- via the *mcast* sessionId whenever possible.

4.1.2.1 Join: connection (branch) setup

The *mcast* user sends the join request to the MCM at the destination to be joined; this is MCM1 in our example. As a result, MCM1 accesses MDS to get the nodeSet (the set of possible sources in the current tree) and the MR to get the route. The route will consist of a sequence of nodes from a selected joining node to destination. If the nodeSet was empty the only possible joining node is the source (it corresponds to a unicast connection between two hosts). In our example the nodeSet was not empty and the joining node selected is MCM2; the route selected is: MCM2-MCM3-MCM1.

The next step is the resource reservation at nodes, process that is coordinated by the MCM at destination. MCM1 contacts the MCMs at nodes in the route: `replicateFlow` will be sent to the joining node (MCM2), while `joinNode` will be sent to the rest of the nodes in route (MCM3). This activity can be accomplished in two ways: sequential setup or parallel setup.

- If sequential setup is used, responses from nodes come back to the MCM1.

1. The term "indication" is used here in the sense of "notification".

- If parallel setup is used, the out vci selected will be sent to the neighbor node in route. Only if the request is rejected the response message is sent to MCM1.

This report focuses on the sequential setup.

Figure 19 shows the sequence of interactions between controllers for the sequential setup.

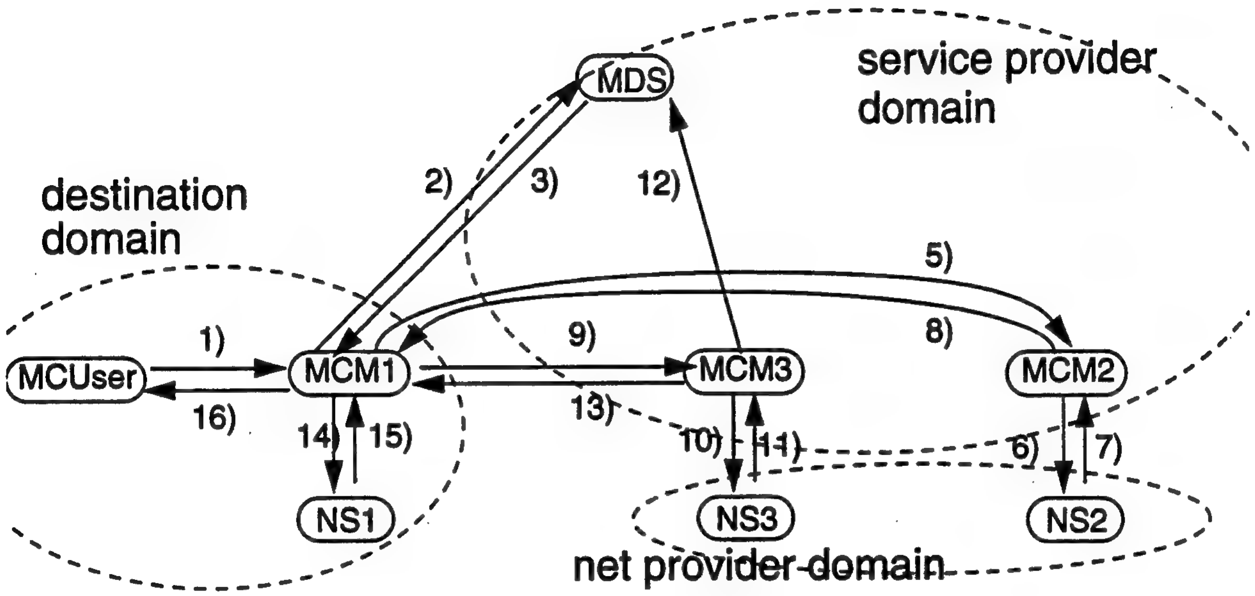


Figure 19

1. joinSessionReq
2. getNodeSetReq
3. getNodeSetRep
4. getRoute (not shown in Figure 19)
5. replicateFlowReq
    6. connectReq (corresponds to a new branch)
    7. connectRep(out-vci)
8. replicateFlowRep(out-vci)
9. joinNodeReq
    10.connectReq (corresponds to a unicast connection)
    11.connectRep(out-vci)

12. addToNodeSetReq
13. joinNodeRep(out-vci)
    14.connectReq (corresponds to a reservation of the input port/vpi/vci)
    15.connectRep
16. JoinSessionRep(input stream interface)

4.1.2.2 Join: connection (branch) undo

If it is the joining node the one rejecting the reservation request, there is nothing to undo.

If it is an intermediate node along the route the one rejecting the request -node 3 in our example-, the resource reservation accomplished up to that point needs to be undone. In our design, the node rejecting the reservation request decides (depending on its state for that *mcast* session) to send or not send a `leavingNode` message up the branch being setup.

This procedure works correctly unless two things happen simultaneously: (1) there is a concurrent `joinNode` request for the same *mcast* session at this node, and (2) NodeServer rejects the first `joinNode` request while accepting the second one. This problem is addressed in Section 2.3.

4.1.2.3 Leave: connection (branch) teardown

Lets assume the same setup as before (Figure 18).

MCM1 gets a request to leave the session. MCM3 should also leave the session because it is not serving other destinations (accurately speaking it is the switch, not the MCM, who is serving destinations). MCM2, on the contrary, will remain in the tree because it is feeding another branch.

The sequence of control messages is shown in Figure 20.

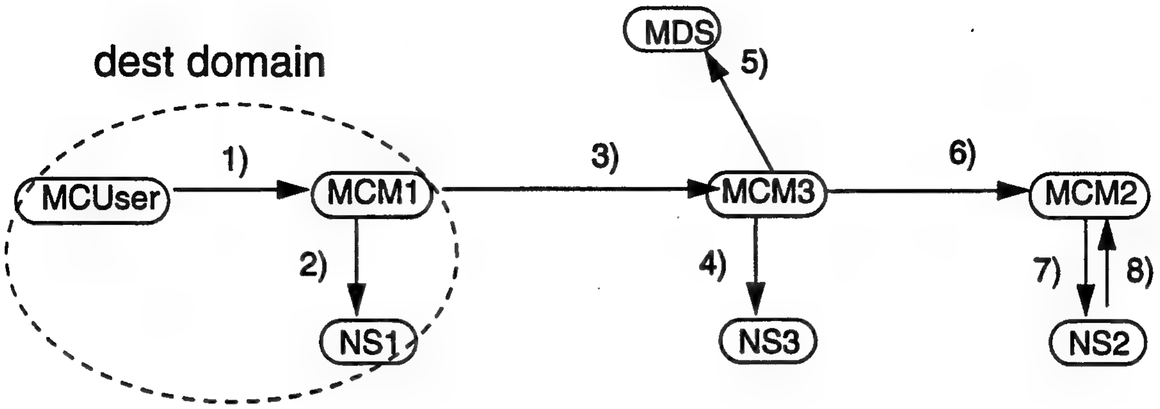


Figure 20

1. leaveSessionReq
   2. disconnectReq (corresponds to a free of the input port/vpi/vci)
3. leavingNodeInd
   4. disconnectReq (corresponds to a unicast connection)
5. removeFromNodeSetReq
6. leavingNodeInd
   7. disconnectReq (corresponds to a branch)

4.1.2.4 Terminate: connection (tree) teardown

MCM at source receives the request `terminate` and forwards the message to all the branches. Let's assume in this scenario that MCM2 is at the source of the tree (Figure 21).

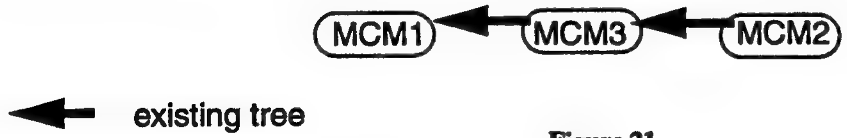


Figure 21

The sequence of operations is shown in next Figure.

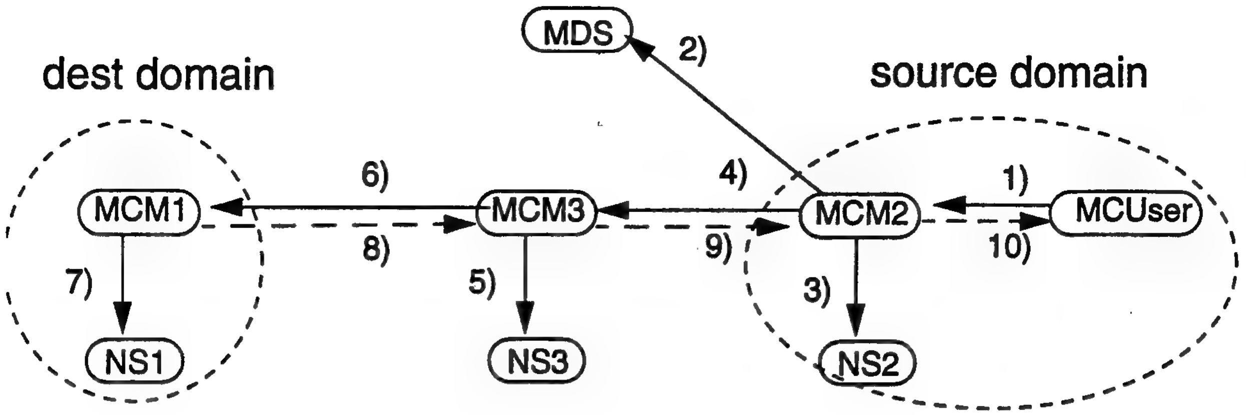


Figure 22

1. terminateReq
2. deregisterSessionReq
    3. disconnectReq
4. terminateReq
    5. disconnectReq
6. terminateReq
    7. disconnectReq
8. (terminateRep)
9. (terminateRep)
10. (terminateRep)

Note that the messages 8, 9 and 10 are sent only if the user wants a reply that acknowledges the end of the session. We introduce a new parameter in the method signature: `terminate(in sessionId, in confirm)` so the user issuing `terminate` can request for acknowledgment that the procedure was successfully completed.

In a larger tree, for example if MCM3 had other branches, it would send `terminate` request messages to each branch and wait until all the `terminate` reply messages are back, before sending its own `terminate` reply message up the tree towards the source.

When MCM at destination (MCM1) receives the `terminate` request it should send a notification to the *mcast* user at that site. We do not address explicitly the *mcast* user interface in this report (as reflected in Figure 14).

4.1.3 *mcast* under concurrent operations

mcast users perform requests to MCMs at the periphery for the same *mcast* session (Figure 23). These requests translate into simultaneous requests on MDS and MCMs all over the network. This concurrency can lead to conflictive situations for the server object and possibly to an incorrect service.

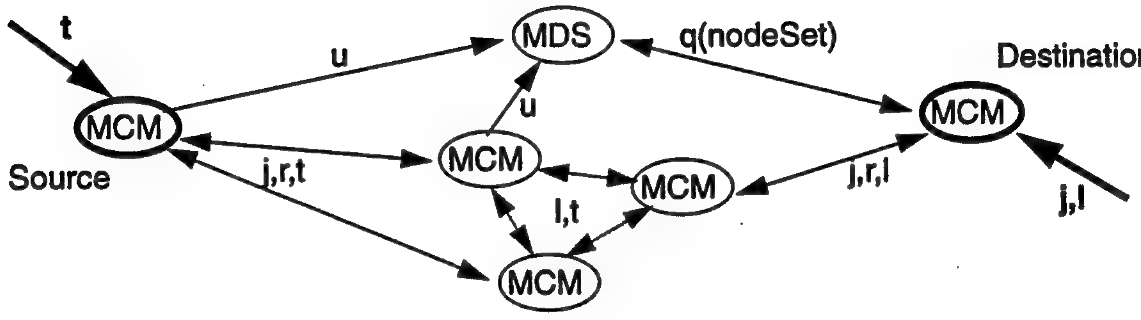


Figure 23

The design of *mcast* described in previous sections realizes the functionality in a system in which, at any time, exactly one service request (per session) is being executed. In this section, we discuss the refinement of the design needed to support service requests to be processed concurrently by *mcast*. Because we assume sessions are independent of each other, for the discussion we focus on a sample session in the system.

The basic problem in a concurrent environment is that there is a chance that the structure of the *mcast* tree will change while a specific service request -the `join` request- is being executed. The following three problem domains have been identified; each leads to a situation that needs to be addressed to ensure the correctness of the service.

- While a `join` request is being processed, the joining node -selected by the router- might leave the tree. As a result, the `join` request is rejected by the mcast service, although the session exists and resources might be available (Section 2.3.1).

- While a `join` request is being processed, one of the nodes in the proposed route is now part of the tree. This results in a pathological tree, in which two edges exist between two nodes. This is a waste of resources we want to avoid (Section 2.3.2).
- While a `join` request is being processed, an edge of the tree is removed, while the `join` request returns successfully. This results in a disconnected destination (Section 2.3.3).

In the following subsections we discuss the above problems and propose refinements of the *mcast* design to address them. As said before we focus on one sample ongoing *mcast* session.

4.1.3.1 Unsuccessful join due to outdated tree information

If by the time the request `replicateFlow` arrives at the joining node, it does not belong to the tree, the request is rejected. In highly dynamic trees this situation leads to long delays in connection setups.

A way to reduce the probability of this event is by limiting the set of possible joining nodes provided by MDS to those with less probability to leave the tree. A simple way to measure this probability is by counting the number of destinations that should leave for the node to leave the tree.

4.1.3.2 Avoiding pathological trees

When several destinations join simultaneously, a switch can receive more than one `joinNode` request for the same session, and the node already belongs to the tree by the time a new `joinNode` request arrives. If we want to avoid a pathological tree (with two edges between two nodes), the node has to react and perform as if a `replicateFlow` message had been received. Furthermore, if the new `joinNode` request shows an `inEp` value different from the one in use, a `leavingNode` indication must be issued in that direction. Figure 24 is used next to show this scenario.

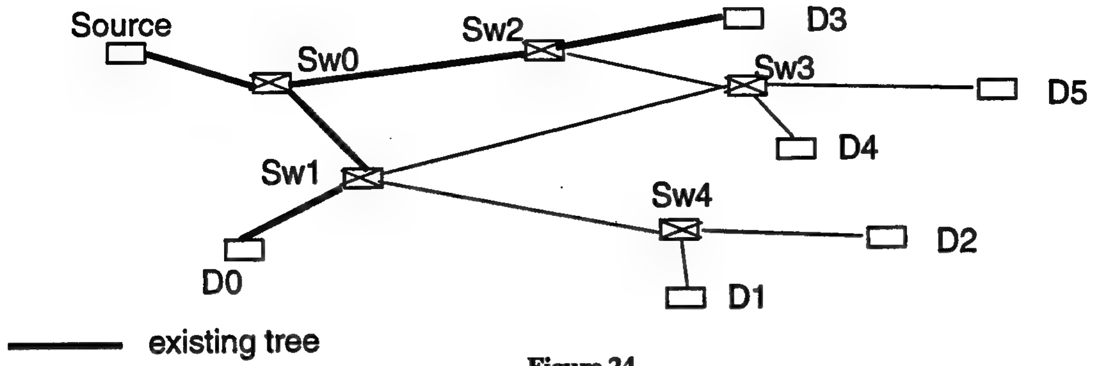


Figure 24

Lets assume D4 and D5 try to join simultaneously.

The router returns the following paths: Sw1-Sw3-D4 and Sw2-Sw3-D5

MCM at Sw3 receives the message `joinNode` from D4. When receiving the same message from D5 it will detect the entry in the table with different inEp (the tree at this point is the one shown in Figure 25). MCM at Sw3 must send `leavingNode` up the tree, in the direction indicated by `inEp` in the request (towards Sw2) and act as in `replicateFlow`.

D5 is not aware of this adaptive behavior.

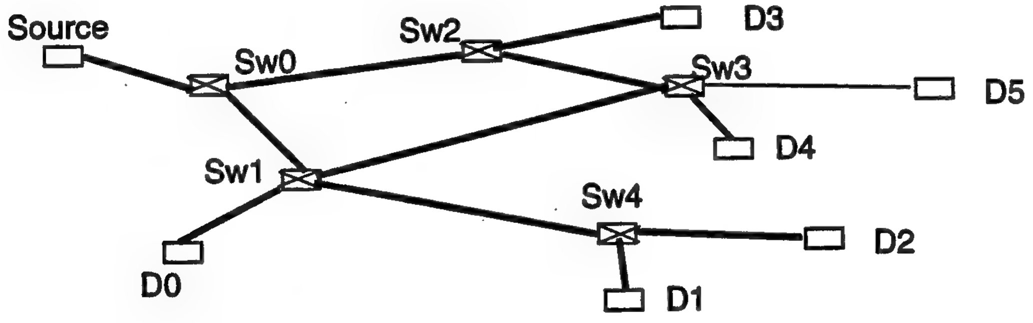


Figure 25

4.1.3.3 Avoiding disconnected trees

Here we give three trace examples in which the tree becomes disconnected. In them a `join` request gets interleaved with (1) a `terminate` (from the source), (2) with a `leavingNode` (from another destination), and (3) with another `join` procedure that is rejected by a *shared* node. The examples are based on Figure 26, which shows an existing *mcast* tree.

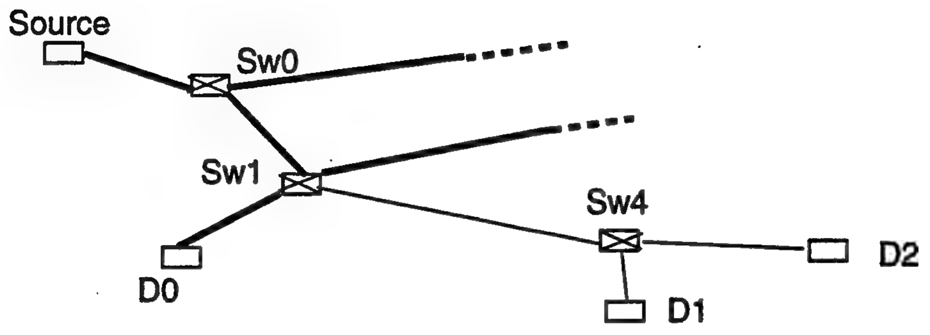


Figure 26

1. The source issues a terminate message while D1 tries to join. Route for D1 is Sw1-Sw4.
   A problematic trace is:

- D1 sends replicateFlow to Sw1. D1 receives a positive reply and sends joinNode to Sw4.
- The terminate message arrives at Sw1 and is forwarded. The terminate message arrives at Sw4 before the joinNode message; because Sw4 does not belong to the tree, the terminate message is ignored.
- Sw4 receives joinNode from D1.

If the joinNode request is not rejected by Sw4 because of a lack of resources, we have a disconnected receiver.

2. D1 and D2 join simultaneously. Both are given the same route: Sw1-Sw4-Dx.
   A problematic (though unfair) trace is:

- D1 sends joinNode to Sw4.
- Sw4 receives a leavingNode message from D2, which is forwarded to Sw1.
- Sw4 receives joinNode from D1.

Again, if the joinNode request is not rejected because of a lack of resources, we have a disconnected receiver.

3. D1 and D2 join simultaneously. Both are given the same route: Sw1-Sw4-Dx.
   A problematic trace is:

- D1 sends joinNode to Sw4.
- Sw4 rejects D1 request and sends a leavingNode message to Sw1.
- D2 sends joinNode to Sw4 and it is accepted.

Given that Sw1 eventually receives a leavingNode message from Sw4--it will stop sending flow in that direction--we have a disconnected receiver (D2).

The design of *mcast* is based on controllers that act upon their local state. The reason why an *mcast* session (a tree) might become disconnected is that local state information of neighbor nodes becomes inconsistent. Our approach to this problem is to introduce a synchronization mechanism into *mcast* which realizes consistency. A detailed design of this synchronization mechanism is not part of this report.

4.2 Making *mcast* manageable

Once we have a stable design for a service, how to make it manageable? By manageable we mean that the service allows its *supervision* by an external management system. Supervision is an activity that relates to the OSI functional areas of performance and configuration management. And because we are managing a service, this activity relates to the functions in the service management layer of the TMN architecture. The management task includes monitoring and controlling the service to ensure that it operates as intended, maintaining the service-level agreements while achieving management objectives.

Specifically, two activities are identified: a) *managing the set of controllers* (i.e. connection managers, etc.) that comprise the functionality of the service delivery system, and b) *managing the set of service instances* (such as audio multicast connections) which are dynamically created, modified and terminated during the operation of the service delivery system. Initially we have focused on b), the management of service sessions.

To make *mcast* manageable we need to modify its design. In which way? Making a system (a service) manageable means making both the system that is being managed and the management system "aware" of each other, by providing addresses at which to communicate.

To make *mcast* manageable, according to the service management model explained in chapter X, (1) the service delivery system has to maintain the service instances. In *mcast,* it is the connection manager (MCM) that acts as service factory when a user issues an `initiate` request. As a result a sessionId is provided back to the user(s). This sessionId is the handle for subsequent operations on the newly created *mcast* session; it represents the user capability to access/control the service instance[1].

Making *mcast* manageable implies the managed system (MCMs) must know about the existence of the management system (the *mcast*SAs and SM, see Chapter X). Figure 27 shows the interaction among MCM and the management system.

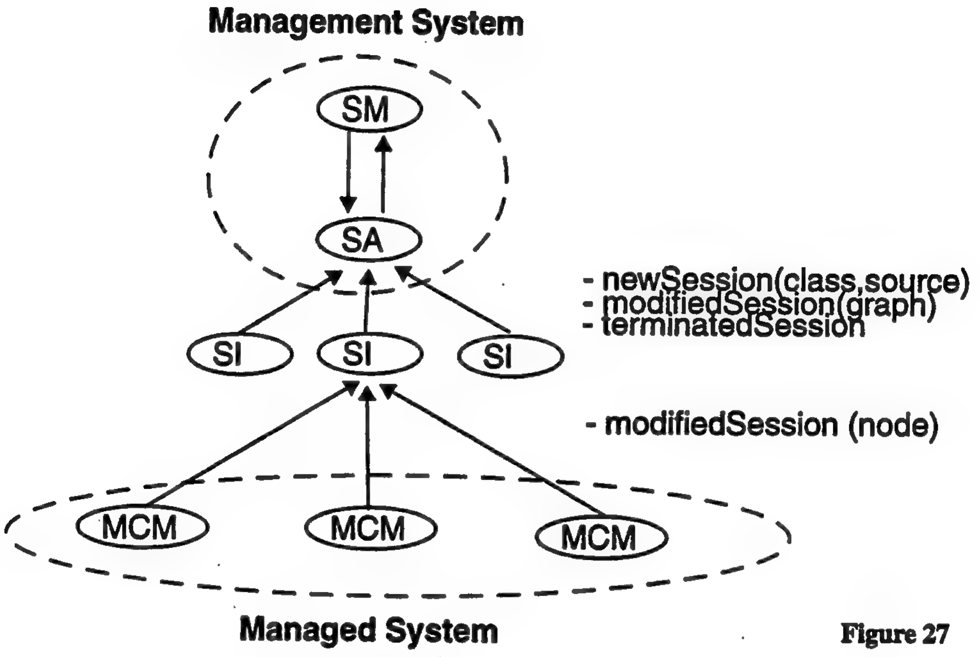


Figure 27

4.3 Implementing *mcast* in Telesoft

1. Actually, the handle is advertised via the MDS.

Telesoft allows to develop common software for different platforms. In our case we are interested in running our system in two different platforms: the emulation environment [CHA96a] and *xbind* [LAZ96], a CORBA-based [OMG96] system. By completely separating the API from the underlying platform we allow the same controller implementations--same sources--to run on any kind of platform that supports distributed objects and asynchronous communication.

The basic structure of an independent-platform object (controller), such as MCM or MDS, consists of two parts:

- the definition of the interface offered, that is, the structure of the messages that can be sent to this object to request (REQ) or notify (IND)[1]. Together with these, we include the structure of the messages to reply to these messages (REP), if any. Note that the latter are, strictly speaking, messages offered at another objects's interface, but its definition is here;
- the behavior of the object, that is, how it responds when receiving a message, whether it is a request received or a reply received that corresponds to a request previously issued to another object.

4.3.1 Multicast Connection Manager (MCM)

The MCM keeps a list of multicast sessions going through its corresponding node. The list contains members of type *MCMcastEntry*:

```
struct MCMcastEntry{
 McastId id;
 ATMEndPointId in_ep;
 ATMEndPointList out_eplist;
 ServiceClassserviceClass;
};
```

An *ATMEndPointId* is a triplet (VPI, VCI, port).

A *McastId* consists of an *ATMEndPointId* and a *HostId* (which is a short). It is unique for each multicast session. The flow of a multicast session gets into a port of a node. That flow is identified by an *ATMEndPointId (in_ep)*. The flow can go on several links to other nodes. These flows are tracked in the *ATMEndPointList out_eplist*. Finally, a multicast session has a well-defined service class, which is also kept in the *MCMcastEntry*.

1. The difference between a request and an indication or notification is qualitative.

The interface offered to the *mcast* user is:

- InitiateSessionReq(ServiceClass class, int localSessionId);
- JoinSessionReq(McastId session, ServiceClass class);
- LeaveSessionReq(McastId session)

The interface offered to the peer MCM is:

- JoinNodeReq(McastId session, ServiceClass class, short outPort, ATMEndPointId inEp);
- JoinNodeRep(McastId session, ATMEndPointId outEp, ReturnCode result);
- ReplicateFlowReq(McastId session, short outPort);
- ReplicateFlowRep(McastId session, ATMEndPointId ep, ReturnCode result);
- LeavingNodeInd(McastId session, MCNodeId nodeId);

The interface offered to both: the *mcast* user and the MCM peer is:

- TerminateSessionReq(McastId session, int confirm);
- TerminateSessionRep(McastId session);

The interface offered to MDS (defined under MDS because it is a reply):

- NodesOfSessionRep(NodeList nodeList, ReturnCode result);

The interface offered to NS (defined under NS because it is a reply):

- connectRep(McastId session, ReturnCode result, out_vci).

Several JoinSessionReq's can be pending in an MCM at the same time. This is because MCMs communicate with each other asynchronously. For each JoinSessionReq that an MCM receives, it creates an instance of the class *JoinState* and adds it to the list *d_joiningList*. Only upon completion of the request (after having communicated with other MCMs), that instance is removed from *d_joiningList* and an answer is given to the user. The same mechanism works for terminating sessions. The sessions about to be terminated are kept in the list *d_terminatingList*.

The interaction with the NodeServer is asynchronous. Because of it, it is possible that, while MCM is waiting for a reply from the NodeServer, reply to a `connect` request, it receives another request (j,r,l,t) on the same *mcast* session. When MCM is waiting for a NS reply its state is undefined because it is not clear if it belongs to the tree or not. Therefore it should not execute any other request on the same session. A mechanism has been implemented to take care of this problem. New requests are stored and their execution delayed until the (same session) NS reply is received.

4.3.2 Multicast Directory Service (MDS)

The MDS contains a table (list of) with *MDSMcastEntry*'s:

```
struct MDSMcastEntry {
 McastId                 id;
 McastDescription        about;
 NodeList                nodes_in_tree;
};
```

The interface offered to the MCM:

- RegisterSessionReq(McastId session, McastDescription description);
- DeregisterSessionReq(McastId session);
- NodesOfSessionReq(McastId session);
- NodesOfSessionRep(NodeList nodeList, ReturnCode result);
- AddToSessionReq(McastId session, long nodeId);
- RemoveFromSessionReq(McastId session, long nodeId);

4.3.3 NodeServer: integration with xbind

The NodeServer has basically two functions: vci management and resource management.

We have implemented our proxy NodeServer with the basic interface *mcast* needs, providing these two management functions. Our proxy NodeServer can run alone in a simulation environment or can be connected to the *xbind* NodeServer. If the latter we map our basic interface into the one defined in *xbind* [BIB, appendix]. Currently the NS interface is as follows.

The interface offered to the MCM is:

- ConnectReq (inEp, outEp, traffic_class, request_id)
- ConnectRep (request_id, result, out_vci)
- DisconnectReq (inEp, outEp, session, traffic_class)

The interface offered to the LAC (Link Admission Controller), only in a simulation environment:

- LACRep (result, request_id)
- LACReleaseRep (result, vci)

The NodeServer at host (AdapterServer) is a special case of NodeServer and the interface to MCM becomes simpler:

- ReserveOutlinkReq (output, traffic_class, request_id)
- ReserveOutlinkRep (request_id, success, out_vci)
- ReserveInlinkReq (input, traffic_class)
- FreeOutlinkReq (output, traffic_class)
- FreeInlinkReq (input, traffic_class)

In the current implementation NodeServer and AdapterServer do not inherit from a common base-class.

There are two issues when thinking of running *mcast* with *xbind*. First, the CORBA environment in which *xbind* is based; second, the functionality that *xbind* provides.

- The Telesoft platform allows to develop software (source code) for different executable environments, one of them being for example a CORBA-based like *xbind*. It is at compilation time that one decides to integrate a specific executable environment.
- There are several aspects concerning the integration of functionality of *mcast* with *xbind*: the Switch Server and Adapter Servers have to be made accessible to our mcast; the VW will access mcast and the transport; finally mcast makes use of the routing. Our proxy NodeServer maps the MCM requests to the *xbind* NodeServer so the real switch (hardware) is reached. Currently our proxy NodeServer is handling the vci management.

5 APIs for Transport

The emergence of distributed multimedia applications exhibiting significantly more stringent QOS requirements than the conventional data-oriented applications call for new transport protocols with different characteristics to co-exist and be integrated within single applications. The different delivery requirements posed by these diverse multimedia applications often imply the need for highly customized protocol implementations. Hence, application developers are faced with the threat of code obsolescence caused by the development of yet newer delivery techniques. For the first time application developers have to contend with the fact that a single protocol stack may perhaps be insufficient to meet all their needs.

In order to address this challenge, we propose a new object-oriented transport architecture in which the atomic entity is based on the consumer/producer paradigm. The architecture consists of (1) consumer/producer components, separately represented by a transport *engine*, and a *front-end* with control and management abstractions, and (2) a set of *binding controllers* implementing Broadband Kernel services such as QOS mapping and dynamic binding of suitable protocol stacks.

The protocol stack is modeled in an object-oriented manner as a sequence of CP components. The protocol stack builder (PSB) is a binding controller responsible for the construction of the protocol stack by binding a variety of consumer/producer engines together. It is aware of the variety of transport components supported on the end-system and the order in which they have to be bound together to create useful protocol stacks. The PSB performs the dynamic binding of the components at run-time and creates a suitable transport protocol stack according to application QOS requirements. The PSB allows to dynamically create a variety of protocol stacks on a per call basis and tailored to the special needs of the application. This differs significantly from the traditional transport architecture that assumes pre-installed transport protocol stacks that cannot be customized.

5.1 Architecture

As said before the architecture consists of (1) consumer/producer components and (2) a set of binding controllers implementing Broadband Kernel services.

A consumer/producer component is represented by two abstractions: the *engine* and the *front-end.*

- The *consumer/producer engine* (CPE) is responsible for data processing (including protocol implementation), buffering and multiplexing. CPEs are located in the data flow path of a multimedia communication session and are the actual hardware and software components that process and transport the data. Their implementation is generally platform dependent.
- The *consumer/producer front-end* (CPF) defines an abstraction layer for the signaling system to remotely control and manage CPEs and the connections established in them.

An example of a CP component in our current architecture is the Transport Protocol (TP) that consists of the couple TP engine and TP front-end (or TP). TP controls the TP engine via a transport protocol *QoS-based engine control interface,* TP qECI. The TP engine can be an implementation of any transport protocol (e.g., **qStack**, TCP, RTP, etc.) as long as it implements the TP qECI. The TP API is provided later.

Binding controllers (BCs) allow the creation, operation, management and programming of services. They elevate the level of abstraction needed to develop multimedia application by encapsulating domain specific knowledge, thus reducing the amount of technical knowledge required by the application developers to write multimedia applications. Examples of such binding controllers are the protocol stack builder (PSB) and the connection manager (CM). The PSB creates, binds and controls media processors and media transporters. The CM creates network connections between end-systems.

Summarizing: these are the three "layers" of transport components: CPEs, CPFs and BCs. The CPEs are located in the data path. The CPFs allow for media control and management abstractions that are separable from their transport interface abstraction. The BCs are located on top and encapsulate domain specific knowledge (e.g., connection management and transport protocol stacks with QOS support).

5.1.1 Consumer/Producer Engines

The functionality of a CPE is data and protocol processing, buffering and multiplexing. The CPE is hardware and operating system dependent. It is the actual hardware and/or software processor that is located in the data path of a multimedia communication.

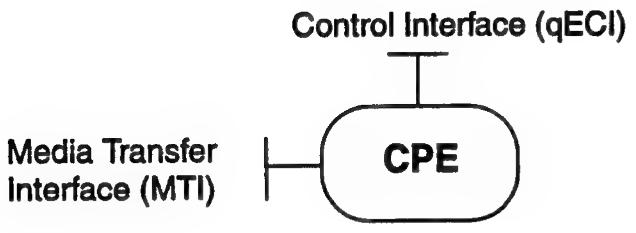


Figure 28 CPE interfaces

The engine has two interfaces: a control interface and a media transfer interface. The interface represented on top of the object is a ***QOS-based engine control interface*** (qECI) visible only to its front-end. In particular, a specific interface is denoted by the component acronym followed by qECI; e.g., the transport protocol QOS-based engine control interface is denoted by TP qECI. This interface is only visible to front-ends that are using the services of the engine.

The interface represented on the side of the object is the engine *media transfer interface* (MTI). It is used to move the data between adjacent CPEs. The interface is only visible to adjacent CPEs in the datapath. Additionally, the MTI provide some i/o control capability to query an engine's maximum service and protocol data unit sizes, set the blocking mode, flush the engine's buffer, etc. An opaque handle (a number identifying the connection) is provided to distinguish between each connection the engine processes. Finally, as opposed to the qECI that is specialized for each CPE, the MTI definition is common to all CPEs so that any CPE can be bounded to any other CPE, and any protocol stack can be build as desired.

When a data unit is transfer via a MTI operation (e.g., via `send()` or `recv()` calls), the CPE processes the data unit and schedules its transfer to an adjacent CPE. Appropriate scheduling and the elimination of unnecessary data copies in the datapath have to be carefully considered to achieve satisfactory overall performance and throughput.

The functionality of a CPE is data and protocol processing, buffering and multiplexing. In fact our architecture distinguishes between two types of CP components that differ in how they process the data in the data path: *media processors* and *media transporters*. Media Processors transform media streams from one format to another by processing the content of the stream. Examples of these include transcoders and encryption devices. Media Transporters carry and route media streams without discerning or altering their contents. The latter implement the transport functionality such as software multiplexing, flow control, segmentation and re-assembly, etc. This chapter wil be focusing on the description of the Media Transporters defined in the current architecture.

CPEs code is introduced in Section 5.4.4 under Implementation aspects.

5.1.2 Consumer/Producer Front-Ends

The CPF defines an abstraction layer for the signaling system to control the heterogeneous transport and computing resources of the end-system. Each CPF abstracts one segment of an end-to-end connection. An end-to-end connection is made of multiple segments, one for each engine it traverses. Examples are each segment of the network connection and each layer of the protocol stack. Abstracting each communication segment allows for each of the connection's protocol stack state machines to be controlled. Multiple front-ends can be attached to a single CPE since CPEs have connection multiplexing capabilities.

The CPFs use dynamically linked libraries that allow them to be bound to the CPEs at runtime. In this way, a protocol stack can be composed and built dynamically. A CPF needs to know only the definition of the type of CPE it abstracts.

The interfaces of the CPFs consist of QOS-based APIs used by the binding controllers and management system to control and manage the CPE connections. They provide capabilities for resource provisioning, accounting and QOS control (e.g., QOS monitoring, QOS violation detection, QOS adaptation and QOS re-negotiation). The interfaces of the CPFs are open CORBA interfaces. Thus, the CPFs allow for end-system transport and computing resources to be remotely controlled.

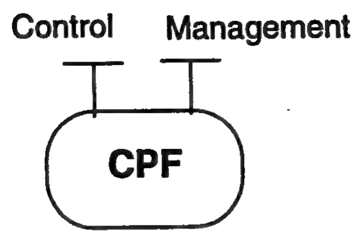


Figure 29 CPF interfaces

Most of the C interface operations will be mapped one-to-one to a qECI operation and the M interface will mainly be used to control and manage the segment connection.

Finally, the set of interfaces is extensible. New services and their corresponding APIs can be added as needed. Moreover, controllers and applications using this architecture don't have to be modified to enjoy this support.

The CPFs APIs defined in the current architecture are presented in Section 5.2.

5.1.3 Consumer/Producer Controllers

As opposed to the CPFs that represent the low-level middleware services, the CPCs (or binding controllers) provide transport services; they represent high-level middleware controllers that an application developer would normally use to build a multimedia application. Examples of such controllers are connection control, transport monitoring, protocol stack building and remote device control. Collectively, these controllers are termed the Broadband Kernel Services (BKS).

Controllers have four interfaces that allow the creation, operation, management and programming of the controller. The interfaces are CORBA open interfaces and allow the controllers to be remotely invoked.

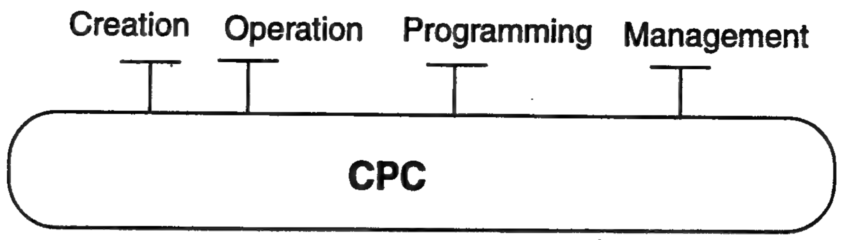


Figure 30 CPC interfaces

- The service invocation interface is the entry point of the execution or instantiation of a service.
- The service operation interface defines the operational functionality of the controller and allows for monitoring and manipulation of service instance states during execution. It is typically the primary interface of the controller.
- The service programming interface allows manipulation of the logic to be performed while a service is in execution.
- Finally, the service management interface allows for monitoring of the controller states and manipulation of controller parameters.

The CPCs APIs defined in the current architecture are presented in Section 5.3.

5.1.4 Interactions between CPEs and CPFs

The purpose of separating the engines from their front-ends is to separate the control of two different time scales. Furthermore, it allows for the remote control of the individual transport connections. Each engine is assumed to have software multiplexing capabilities and may support multiple connections that are each abstracted by a single front-end that contains the specific state of the connection. The state of a connection is defined as the required QOS and the measured QOS. Engine assigns an opaque handle to access the state and provide the handle to the front-end. All front-end that are bound to an engine access the capabilities of the engine via its qECI and the connection handle.

Figure 31 illustrates the interaction between five CPEs. The CPE under consideration is represented by the oval in the middle and is bounded by four CPEs (a through d). The CPE multiplexes three connections (id1 to id3). The segment of each connection at this CPE is represented by a CPF bounded to it (1 to 3). CPE-c and d could be an AAL5 media transporter and a UDP media transporter while the CPE under consideration could be **qStack** transport protocol engine, and the CPE-a and b could be an audio and video streamed device. This illustration would represent two audio communications, one going though an IP network and one through an ATM network, and one video communication that would go through an ATM network. In our example, **qStack**, the middle CPE is an implementation of real-time interactive transport protocol with QOS support.

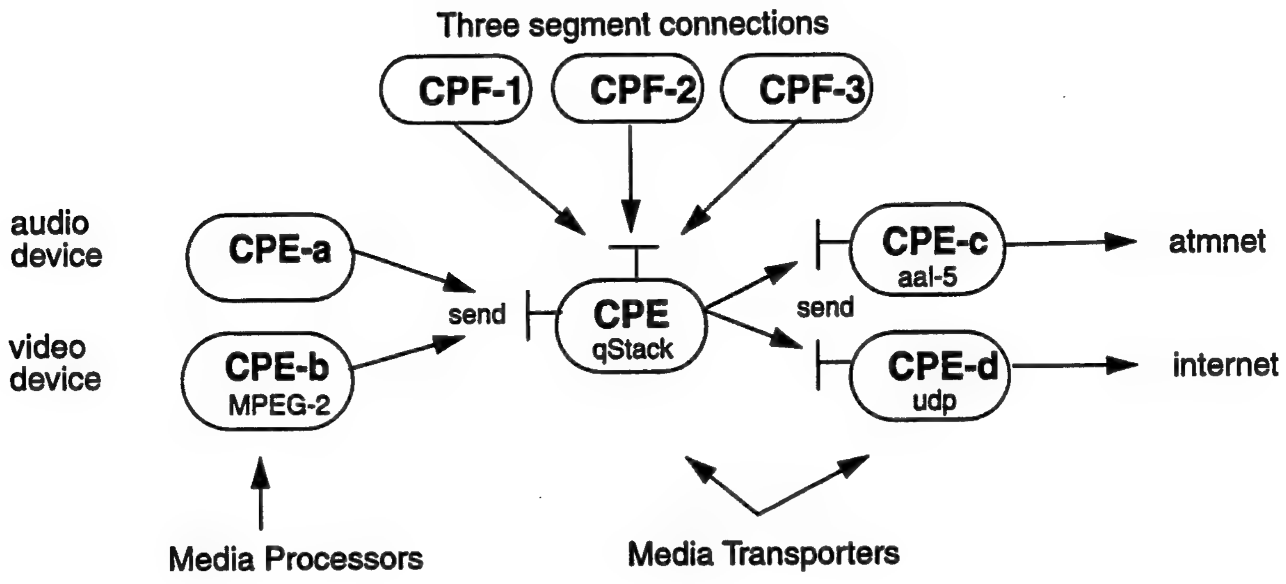


Figure 31 Example of protocol stacks for three specific connections

Figure 32 shows a more detailed illustration that includes all the existing CPFs: one CPF attached to CPE-B to control the video stream, one CPF attached to CPE-D to control the UDP/IP connection, one or two CPFs attached to CPE-C, depending upon whether or not they go to the same destination on the same network channel (in which case channel multiplexing would be performed in CPE-C). Finally, there would be one or two CPFs attached to the CPE-A, depending whether there is one audio communication multicast in software or two audio communications mixed and controlled at the CPE-A.

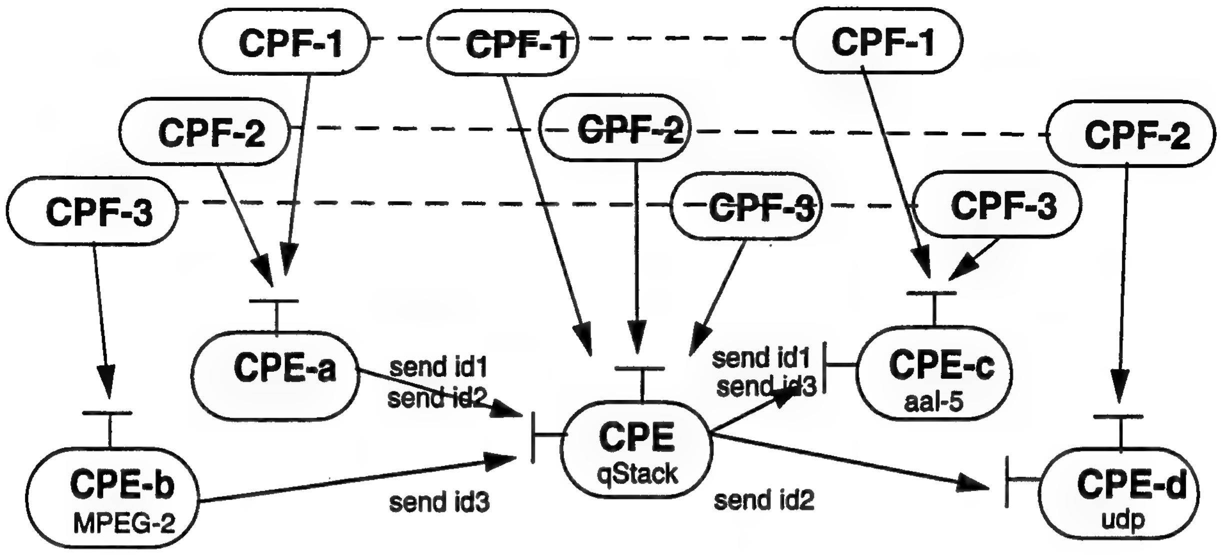


Figure 32 Existing CPFs for three specific connections

5.2 Media Transporter Components

As mentioned in Section 5.1.1, there are two main consumer/producer components in our transport architecture: *media processors* and *media transporters*. The current architecture defines two media processors, namely (Stream) that abstracts media stream producers and consumers, and (Encrypt), and three media transporter components, namely the multiplexer port (MuxP), the transport protocol (TP) and the network provisioning (NetP).

The media stream producers grab data via a physical device (e.g., camera and microphone) and encodes the raw data into compressed data (e.g., a video board that compresses raw video into MPEG-2 encoded video). The media stream consumers render the data to a physical device (e.g., display and speaker) after having decoded the compressed data (e.g., a DSP chip set that processes encoded audio and play it back). Finally, the encryptor either encrypts or decrypts data depending whether it is located on the sender or receiver side of the communication.

In the rest of the Chapter we will focus on the media transporters.

Media transporters carry and route media streams without discerning or altering their contents. They implement the transport functionality such as software multiplexing, flow control, encapsulation and de-encapsulation, segmentation and re-assembly, etc. On the transmitting side of a connection, media transporters may add a header to each data unit that can be used to communicate with the receiver peer. The transmitting side may also fragment the message if it is too large to be transferred to the following media transporter in the protocol stack. On the receiving side, media transporters reassemble messages, de-encapsulate and forward data units to the next component in the protocol stack. A chain of media transporter is traversed until the data reaches a media processor such as a streamed device to render it back or a saves it in a file for future use.

In this section we describe three specific media transport components to be located in the datapath: the network provisioning protocol (NetP), the transport protocol (TP) and the multiplexer (MuxP). In Section 5.2.4 we will provide the CPF APIs, while the CPEs specifics are in Section 5.4.4 under Implementation aspects.

5.2.1 Network Provisioning (NetP)

The network provisioning (NetP) component of the protocol stack is responsible for data transmission across networks. It provides transparent transfer of the data for the transport protocol component. NetP provides the lowest level of the protocol stack on the end-system. It is the base layer of the end-system protocol stack.

The processing capabilities of the NetP CPE are related to the per-byte (or per-bit) operations (e.g., checksum). The CPE is expected to deliver error free packets, but could potentially deliver errored packet with a notification to its recipient. No assumption is made about the ordering of delivered packets. The media transfer interfaces simply remap the calls to the equivalent system or library calls. Examples of protocols that the NetP CPE provides are UDP/IP and AAL5/ATM.

The NetP CPE API allows for the network provision of calls and to potentially pace the injection of packets into the network. The provisioning capabilities are the creation and destruction of opaque network handles (e.g., sockets or file descriptors) associated with network endpoints. Its QOS control capabilities are related to the pacing of packet injection into the network and the monitoring of the packet QOS. The accounting capability is the counting of the number of packets transferred. The control capabilities are typically realized by invoking system or library calls that communicate with a network interface card driver. For the pacing capability, interval timers might be used. However, pacing is expected to be provided at the driver level or network interface card since software pacing is not efficient.

The NetP CPF control and management functionality is limited to opening and closing network connections and setting network QOS requirements when applicable. The main capability of the front-end is to load dynamically a CPE and to obtain the engine MTI and identifier of the connection it abstracts. The control capabilities consist of the provisioning of calls (e.g., opening/closing virtual circuits) and network QOS control such as pacing the injection of packets into the network. The management capability is related to accounting. The front-end do not perform directly the control and management capabilities but maps calls to its engine, marshalling the arguments of the calls whenever needed.

5.2.2 Transport Protocol (TP)

The transport protocol component of the protocol stack is used to provide the end-to-end communication capabilities. It is the first layer in the protocol stack that is end-to-end QOS-aware. If needed, TP must have the ability to ensure reliable end-to-end communication. In essence, TP relies on NetP to received error free data, but may have to request for retransmission if a segment of data is missing. Flow control is included in this layer to manage the data flow through the connection. This means that TP assumes that a feedback channel is available for QOS adaptation and flow control.

The TP CPE functionality is to ensure efficient delivery of data and to perform the ``in-flow" (or fast time scale) QOS monitoring. It ensures that end-to-end QOS is provided to the next CPE bound to it. The TP CPE implements the flow control, encapsulation/decapsulation, segmentation and re-assembly, buffering, multicasting and QOS adaptation mechanisms. The TP CPE assumes that for each connection, it will be bound to a NetP CPE.

The TP CPF defines the end-to-end QOS control, management and accounting capabilities. Its control interface contains a binding method so that a NetP CPE can be bound to the TP CPE. The end-to-end QOS control API is for setting the parameters of the transport protocol and for monitoring the QOS delivered to a specific connection. The specific capabilities of the control interface depend on the transport protocol specifications. In particular, it can be used to set end-to-end QOS to be delivered to the application. One of the management interface capabilities is the selection of the transport protocol to be used for the connection. It can also be used to obtain accounting information such as the call duration and the amount of recovered data delivered to the application. The TP CPF also performs the slow time scale monitoring of connections and initiates QOS re-negotiation upon detection of sustained QOS violations. Finally, the TP shall have the capability to change its associated CPE dynamically to adapt to large QOS variation requiring additional capabilities to be kicked in automatically.

5.2.3 Transport Multiplexer (TMux)

TMux provides the capability of multiplexing of connections into one TP connection when multiple media processors of a single multimedia application are simultaneously active (e.g., DMIF FlexMux). This capability is also useful for connection where many short terms connections are opened and closed. A web browsing represents such type of application. For example, a client could establish a connection with a server, and many short transactions are executed within the unique network connection. This relieves the system of the overhead of connection establishment for short transactions such as commonly done in web browsing.

The TMux CPF provides the ability to add and destroy TMux channels with a given QOS and ensure that the overall QOS budget of the connection is within the negotiated QOS of the TP that it is bound to. The TMux CPE simply performs the software multiplexing of the TMux channel onto the port it is bound to.

5.2.4 QOS-based APIs

It is proposed that every CPF be a specialization of an interface referred to as a VirtualPort. The main reason for VirtualPort definition is to allow CPF object reference to be passed transparently, no matter of which specific type the CPF actually implements. Second, since CPF object references are used for binding, there is a common set of operations and attributes required.

The VirtualPort interface allows any transport component in the data path to be referred to as a VirtualPort, independently of its specialized control and management capabilities. The obvious advantages of this approach is the flexibility it provides for constructing protocol stacks transparently and for avoiding modifying the CPF s every time a component's interface is modified or a new component added.

For all methods, a {Reject} exception can be raised to indicate that a provided parameter is not valid or that the execution of the method failed.

5.2.4.1 VirtualPort: a common CPF API for binding CPEs

```
enum PortDirection {
   InPort, OutPort, BiPort
};

enum PortConnection {
   UnicastPort, MulticastPort, MultiplexPort, SwitchPort
};
enum PortEvent {
   PortError, PortClosed, PortDataReady, PortAsyncRecv
};
typedef long PortBindingPtr;  // local binding pointer
                              // to be cast appropriately
interface TFactory;           // pre-declaration
interface TMonitor;           // pre-declaration
interface VirtualPort;        // pre-declaration

typedef sequence<VirtualPort> PortList;

interface  VirtualPort
{
   readonly attribute   string         type;
   readonly attribute   PortDirection  direction;
   readonly attribute   PortConnection connection;
   readonly attribute   PortList       attachedPorts;
   readonly attribute   TFactory       factory;
   readonly attribute   TMonitor       monitor;
   readonly attribute   PortBindingPtr CidPtr;
   readonly attribute   PortBindingPtr MTIPtr;

   boolean  setAttributes ( in PortDirection direction,
                            in PortConnection connection,
                            in TFactory factory,
                            in TMonitor monitor )  raises (Reject);

   boolean  attachPort    ( in VirtualPort port )  raises (Reject);
   boolean  detachPort    ( in VirtualPort port )  raises (Reject);

   boolean  setCallback   ( in BindingPtr callback,
                            in PortEvent event )   raises (Reject);
};
```

Attributes

- [type] The type refers to the name of specialized interface of the port. Examples of values are VirtualNetP, VirtualTP and VirtualMuxP.
- [direction] The {direction} refers to the direction of flow of data defined from the application to the network. An {InPort} direction means that the data units are incoming from the network; that is, data units flow from the network toward the application. An {OutPort}

value means the data units flow from the application towards the network. Finally, a {BiPort} value means that the flow of data is bi-directional; that is, the port can be used to send and receive data. The direction is bi-directional by default.

- [connection] The {connection} refers to the multiplexing capabilities of the port. By default, the port is a {UnicastPort}, that is, it provides a one-to-one mapping between its input and output. A {UnicastPort} can take any direction value. A {MulticastPort} {must} be of direction {OutPort}and can have multiple {OutPort} attached to it. It is not allowed to have any {InPort} and {BiPort} attached to it.For example, an {OutPort} TP could have two {OutPort} NetP attached to it (IP and ATM) and thus would multicast the data over an IP and ATM network.
- A {MultiplexPort} can be of either {InPort} or {OutPort} direction. It can have only one VirtualPort of the same direction attached to it, but multiple VirtualPort can have it attached to them. For example, in the case of {OutPort}, it performs multiplexing of several channels onto a single {OutPort} channel. For an {InPort} direction, it does de-multiplexing of a single {InPort} channel into multiple higher level VirtualPort CPE or media processor CPE via callback (see below for description of {setCallback} method). Finally, a {SwitchPort} can have multiple of both {InPort} and {OutPort} attached to it. It perform software switching in the transport layer.
- [attachedPorts] The {attachedPorts} contains the list of all VirtualPort which that been attached to the VirtualPort with the {attachPort} method described below.
- [factory] The {factory} is a pointer to the transport factory that created the object.
- [monitor] The {monitor} is a pointer to the PortMonitor with whom the VirtualPort is registered. It can have a nil value if the port has not been registered with any transport monitoring CPC.

The last two attributes refers to binding pointers related to the MTI. The MTI is implemented as a dispatch table and specified in the C programming language. The {CidPtr} and {MTIPtr} attributes are pointers to the actual attributes and are provided as {PortBindingPtr} which is of type {long}. The pointers have to be cast back to the appropriate type in the engines. The usage of these attributes should be local.

- [CidPtr] The value pointed by the attribute {CidPtr} refers to the opaque handle to the engine connection that this port front-end abstracts. The {cid} is set when the segment of the connection is set in the engine and provided by the engine. The type to which the attribute should be locally cast is specified in the CPE \qECI. A {cid} value of -1 indicates that the connection is not setup.
- [MTIPtr] The value pointed by the attribute {MTIPtr} refers to the MTI dispatch table located in the CPE. See the CPE API in Section 5.4.4.

Methods

- [setAttributes] This method should be invoked by the PSB to set all attributes of the VirtualPort. The remaining attributes ({type} and binding pointers) are to be set by the PortFactory at creation time.
- [attachPort] This method is used to bind a CPF to the VirtualPort. When a VirtualPort is attached, its reference is put in the list of attached ports.

VirtualPort are attached to their adjacent upper layer, that is, in the case of a data path sequence of StreamP--TP--NetP, the NetP will be attached to the TP by invoking TP's {attachPort} method with NetP as argument. Then TP will be bound to the {Stream} by calling the {Stream}'s {attachPort} method with TP as argument.

- [detachPort] This method is used to unbind a CPF to the VirtualPort.
- [setCallback] The {setCallback} method is used to provide a callback function to the CPE. Four types of callback function can be specified by providing the event type as an attribute to the method. The prototypes of the callback methods are specified in the CPE qECI definition. The callback pointer is to be passed down to the CPE that will implement the callback and make the call upon occurrence of the event.

5.2.4.2 VirtualNetP: Network Provisioning CPF API

The NetP is a kind of VirtualPort. Its control and management interface abstracts the network provisioning capability. The NetP is a specialized VirtualPort. By default, it is expected to be have a value of {BiPort} as direction and a value of {UnicastPort} for connection type. It is not expected that any port will be attached to a NetP. Finally, it is not expected that a monitor will be monitoring a NetP except in the case where the NetP is the only transport component instantiated by the PSB.

```
#include "VirtualResource.idl"
#include "VirtualPort.idl"
#include "QOS.idl"
#include "EndPoint.idl"

interface VirtualNetP : VirtualPort, VirtualResource
{
   readonly attribute   NetworkQOSSpec QOSSpec;
   readonly attribute   EndPoint       EP;

   boolean  open        ( in NetworkQOSSpec qos,
                          in EndPoint ep )       raises (Reject);
   boolean  close       ( )                      raises (Reject);
```

```
        boolean  setQOS      ( in NetworkQOSSpec qos ) raises (Reject);
        NetworkQOSProfile queryQOS ( )                 raises (Reject);
    };
```

Attributes

The two readonly attributes, {QOSSpec} and {EP}, are set when the {open} method is invoked.

- [QOSSpec] When the NetP is an {InPort}, the {QOSSpec} specifies the QOS the NetP is expected to receive from the network and deliver to the next component in the datapath. When it is an {OutPort}, the attribute contains the traffic description it must use for its controlling its pacing mechanism (e.g., leaky bucket). The {QOSSpec} parameter can be changed at run-time using the {setQOS} operation.
- [EP] The {EP} parameter contains the appropriate endpoint information for the network handle to be created. For examples, in the case of an ATM network endpoint, it can be a virtual path and virtual channel identifier pair, and, for an IP network, an IP address and IP port pair.

Methods

- [open] The {open} method is used to create the opaque network handle. It must also set the value of the {cid} appropriately.
- [close] The method {close} is used to terminate the connection and destroy the network handle.
- [setQOS] The {setQOS} method allows to change the QOS attribute at run-time, after the connection has been opened. This method should be invoked after a successful renegotiation operation. On the sender side, the QOS specifies the new traffic description, and on the receiver side, the renegotiated QOS parameters.
- [queryQOS] The {queryQOS} method allows for retrieving the monitored QOS; that is, the QOS delivered by the network.

5.2.4.3 VirtualTP: Transport Protocol CPF API

The TP is a kind of VirtualPort. It should be the first component in the datapath aware of the end-to-end QOS concept. Its control and management interface abstract the transport protocol component of the conventional transport layer. Its interface is defined in virtualtp. By default, TP has a value of {BiPort} as direction and a value of {UnicastPort} for connection type. When TP's direction is set to {BiPort}, it assumes to be bound to a {BiPort} NetP. For reliable connection that requires a feedback channel, TP might be unidirectional if no in-band signalling is used by the engine, or it can support bi-directional connections if the engine demultiplexes the in-band signal messages from the data. The type of transport protocol the engine implements is provided by the VirtualPort::connection attribute.

Finally, TP is expected to be registered with a transport monitor server.

```
#include "VirtualResource.idl"
#include "VirtualPort.idl"
#include "QOS.idl"

interface VirtualTP : VirtualPort, VirtualResource
{
   readonly attribute  TransportQOSSpec          QOSSpec;

   boolean establish  ( in TransportQOSSpec qos ) raises (Reject);
   boolean release    ( )                         raises (Reject);

   boolean setQOS     ( in TransportQOSSpec qos ) raises (Reject);
   TransportQOSProfile queryQOS( )                raises (Reject);

   boolean changeEngine ( in string description ) raises (Reject);
};
```

Attributes

[QOSSpec] The {QOSSpec} parameter specifies the end-to-end QOS to be delivered to the application. It is set when the {establish} or {setQOS} methods are invoked.

Methods

- [establish] The {establish} method initiates a logical end-to-end connection. It may initiate the handshaking, clock synchronization (for QOS monitoring of delays), sequence number negotiation, etc.
- [release] The {release} method terminates the connection and releases all the transport protocol resources associated to it.
- [setQOS] The {setQOS} method allows to change the {QOSSpec} attribute at run-time, after a connection has been established. This method should be invoked after a successful renegotiation operation was performed. On the sender side, the QOS specifies the new traffic description, and on the receiver side, the renegotiated QOS parameters.
- [queryQOS] The {queryQOS} method allows for retrieving the monitored end-to-end QOS; that is, the QOS delivered by the transport to the application.
- [changeEngine] The {changeEngine} method allows for changing of the transport protocol engine dynamically. The argument is the new engine description as is to be provided to the factory at creation time.

5.2.4.4 VirtualMuxP: Multiplexer CPF API

```
#include "VirtualResource.idl"
#include "VirtualPort.idl"
#include "VirtualCapacityRegion.idl"
#include "QOS.idl"

typedef VirtualCapacityRegion VCR;
typedef short TMuxCid;

interface VirtualTMuxP : VirtualPort, VirtualResource
{
   readonly attribute    VCR   initialCapacityRegion;
   readonly attribute    VCR   availableCapacityRegion;

   readonly attribute    long  nChannels;

   boolean setVCR      ( in VCR vcr )              raises (Reject);

   TMuxCid newChannel   ( in TransportQOSSpec qos ) raises (Reject);
   boolean deleteChannel(in TMuxCid cid )          raises (Reject);

   boolean joinChannel ( in TMuxCid cid,
                         in TransportQOSSpec qos
                         in long callback )        raises (Reject);
   boolean leaveChannel( in TMuxCid cid )          raises (Reject);

   boolean setQOS      ( in TMuxCid cid,
                         in TransportQOSSpec qos)  raises (Reject);
   TransportQOSSpec    getQOSSpec (in TMuxCid cid) raises (Reject);
   TransportQOSProfile queryQOS  (in TMuxCid cid) raises (Reject);
};
```

Attributes

- [initialCapacityRegion] This {initialCapacityRegion} attribute indicate the initial multimedia capacity region that was associated with the multiplexer.
- [availableCapacityRegion] The {availableCapacityRegion} attribute indicate the available multimedia capacity region left from the initial capacity region that was allocated.
- [nChannels] The {nChannels} attribute indicates how many channels are currently established in the MuxP.

Methods

- [setVCR] The {setVCR} method assigns the multimedia capacity region of the multiplexer. A call reducing the allocated capacity region must support all resource allocated otherwise it will fail.

- [newChannel] The {newChannel} method initiates a logical connection. The call returns the identifier of the channel to be used at the media transfer interface. This call is to be performed only on an VirtualPort with an {OutPort} direction.
- [deleteChannel] The {deleteChannel} method releases the multiplexer resources and QOS associated to the channel created using the {newChannel} method.
- [joinChannel] This call is similar to the {newChannel} call, except that it is to be performed on a VirtualPort with an {InPort} direction. The {TMuxCid} argument is the identifier of the channel that was created on the sender side. The callback argument is associated to a {PortAsyncRecv} event. When the multiplexer receives a packet for on that channel, it invokes the callback with the received data.
- [leaveChannel] The {leaveChannel} method releases the multiplexer resources and QOS associated to the channel created using the {joinChannel} method.
- [setQOS] The {setQOS} method allows to change the {QOSSpec} attribute to a channel at run-time. On the sender side ({OutPort}), the QOS specifies the new traffic description, and on the receiver side ({InPort}), the renegotiated QOS parameters.
- [getQOSSpec] The {getQOSSpec} method specifies the end-to-end QOS to be delivered to the application. The QOS retrieve is the one set when the channel is created or from the {setQOS} method invocation.
- [queryQOS] The {queryQOS} method allows for retrieving the monitored end-to-end QOS on the identified channel; that is, the QOS delivered by the multiplexer to the application.

5.2.4.5 PortFactory: CPF Factory API

```
#include "VirtualPort.idl"
#include "VirtualNetP.idl"
#include "VirtualTP.idl"
#include "VirtualMuxP.idl"

interface PortFactory
{
   readonly attribute PortList    Ports;

 VirtualNetP    createNetP  (in string descr) raises (Reject);
 VirtualTP      createTP    (in string descr) raises (Reject);
 VirtualTMuxP    createTMuxP (in string descr)  raises (Reject);

 boolean        destroyPort (in VirtualPort port) raises (Reject);

 PortBindingPtr getCidBindingPtr (in long index) raises (Reject);
 PortBindingPtr getMTIBindingPtr (in long index) raises (Reject);
 boolean        setBindingPtr  (in long index,
                                 in PortBindingPtr pCID,
                              in PortBindingPtr pMTI )raises (Reject);
};
```

Attributes

- [Ports] This attribute is a sequence of all VirtualPort that were created with this factory.

Methods

- [createXXX] These factory methods are self-explanatory. The parameter {description} passed as an argument characterizes the engine to be associated with the front-end. An exception can be raised when the creation of an object is created if the engine to be associated to the VirtualPort is not found or at destruction time in the event of an unknown type of port (e.g., a media processor port) is tried to be destroyed.
- [getXXXBindingPtr] The two methods enables the retrieval of the binding pointers values by the VirtualPort. The index is provided to the VirtualPort at creation time by the factory.
- [setBindingPtr] This methods allows to set the binding pointers values after an engine connection has been setup.

5.3 Transport Services

Here we describe the three proposed transport services to be provided by the broadband kernel: the protocol stack builder, a QOS mapper that performs translation of QOS between different protocol stack layers and a monitoring server that performs accounting for management purposes.

The protocol stack builder creates protocol stacks by invoking the control and management interfaces of the NetP, TP and TMux,. The transport monitor performs accounting for management purposes (i.e., the monitoring of the amount and kind of data transferred and the connection holding time). Finally, the QOS mapper performs translation between QOS specifications of different protocol stack layers.

5.3.1 Protocol Stack Builder

The protocol stack is modeled in an object-oriented manner. The *protocol stack builder* (PSB) is responsible of constructing the transport section of the datapath by binding TMux, TP and NetP CPFs together. The PSB can be a centralized server, but it is expected highly distributed (one on each end-system). The PSB is expected to be aware of the variety of engines supported on the end-systems and the order in which they have to be bound to create a useful protocol stack. The PSB performs the dynamic binding of CPEs at run-time by attaching CPF ports. Its functionality is similar to the one of a connection manager that creates a virtual channel in an ATM network, but does so on the end-system.

```
#include "VirtualPort.idl"
#include "EndPoint.idl"
#include "TMonitor.idl"

interface PSB
{
    VirtualPort create ( in EndPoint ep,
                         in PortDirection  direction,
                         in PortConnection connection,
                         in TransportQOSSpec qos ) raises (Reject);
    boolean destroy    ( in VirtualPort port )     raises (Reject);
    boolean join       ( in VirtualPort port,
                         in EndPoint ep)           raises (Reject);
    boolean leave      ( in VirtualPort port,
                         in EndPoint ep)           raises (Reject);
    boolean setQOS     ( in VirtualPort port,
                         in TransportQOSSpec qos ) raises (Reject);
};
```

Methods

- [create] The {create} method is used to request the protocol stack builder to create a protocol stack satisfying the set of parameters passed as argument. Based on the requested QOS, it initializes a network provisioning CPF and possibly a transport protocol and transport multiplexer CPF. It binds the transport components together before returning to its requestor the VirtualPort CPF base object reference to be provided to the application layer.
- [destroy] The {destroy} method releases all the resources associated with a protocol stack instance.
- [join] This method allows to add a new endpoint to the TP CPF, while the a connection is active, creating a multicasting session as at least two endpoints will be attach to the transport once this method returns. This method can also be used to change an endpoint after a QOS renegotiation phase. For example it provides the capability to switch from one network to another (e.g., from IP to ATM) dynamically.
- [leave] This release the resources associated to an endpoint attached to the transport using the {join} method.
- [setQOS] The {setQOS} methods permits to change the end-to-end QOS of the TP. The network QOS must be changed appropriately by the PSB.

5.3.2 Transport Monitor

The *transport monitor* (TMon) is used to log accounting information for management purposes. It interface is used by the CPF s to register connections. When the datapath comprises a TP, then only the TP registers. When no TP is used, then the NetP register its connection. When a CPF un-registers its connection the accounting information is logged permanently for future usage by the management system (e.g., for billing, network performance and dimensioning).

Any type of information related to the amount of data transferred, the duration of a call, etc., can be logged. Furthermore, the received and expected QOS can be logged. Finally, whenever QOS re-negotiation is successfully performed with a TP, the new QOS is logged.

```
#include "VirtualPort.idl"
#include "QOS.idl"

interface PortMonitor
{
    short       register   (in VirtualPort port,
                            in QOSProfile qos)  raises (Reject);
    onwway void unregister (in short portID);
    onwway void setQOS     (in short portID, in QOSProfile qos);
    onwway void fwdQOSMeas (in short portID, in QOSProfile qos);
};
```

Methods

- [register] The {register} should be used by a transport protocol CPF to register itself with the monitor when it is initialized. This {register} operation returns a ({portID}) identifier to be used in all subsequent operations. It may raise a {Reject} exception to indicate that it is not able to register the port.
- [unregister] The operation {unregister} close the accounting log for the connection.
- [setQOS] The {setQOS} operation is used to change the connection's QOS requirement at run-time.
- [fwdQOS] The {fwdQOS} method is used to instruct the monitor of the measured QOS.

5.3.3 QOS Mapper

The *QOS mapper* (QOSMapper) performs translation of QOS specifications between the various protocol stack levels of QOS (application, transport and network). Unlike the protocol stack builder that controls various CPF, the QOS mapper does not interact with them. The mapping capability is invoked by applications or PSBs to perform QOS translation at call establishment and QOS re-negotiation time..

```
#include "QOS.idl"

interface QOSMap
{
   boolean map ( in    QOSSpecification from,
                 inout QOSSpecification to ) raises  (Reject);
};
```

Methods

[map] The interface consists of this single method. {map} translates the QOS specification provided by the parameter {from} to the QOS level specified by the member {to.QOSLevel} of the second parameter. It returns its translation via the inout parameter {to}. It may raise a {Reject} exception to indicate that it is not able to perform the mapping requested.

5.4 Implementation Considerations

This section details the interaction mechanisms between CPEs and CPFs and presents some implementation considerations.

5.4.1 Implementation Model

The fundamental idea in separating the CPE and CPF implementations is to avoid re-implementing the CPF by each CPE provider and to ensure that the CPF functionality is properly implemented. It is similar in philosophy to the windows open software architecture, where Microsoft specifies both the API and SPI, and implements the middleware between the API and SPI. As long as the service providers satisfy the SPI, any application using the API will run.

Another important idea, is that the CPE are located in the datapath, and therefore, must be optimized for speed, and memory usage. Finally, by having the separation of implementation between CPE and CPF, multiple software providers can implement their own CPE, and the CPF can change, without having to have the CPE providers to re-implement their CPE. This is very important for efficient software development as two separate group can work in parallel into the development of the transport components without needing the constant cooperation of the other group.

For example, during xbind 2.0 development, the specification of the transport components' CPE and CPF interfaces allowed the applications to be developed using preliminary implementation of the CPE and CPF. If a monolithic approach would have been used, then the application development would have been required to wait until all the transport component being available before allowing the proper development. Also, having dynamic loading of CPE, avoid recompiling all the interfaces every time a detail is changed in the CPE.

In fact the CPE are the components that require the most development work as they need to have an efficient implementation and necessitate re-compilation very often. Furthermore, they might be several implementation of CPE for a single CPF. De-coupling their implementation avoid recompiling the same code over and over.

The CPEs are implemented as dynamic libraries and are located in the same address space as the CPFs. The CPE functionality is invoked via a qECI procedure dispatch table. The CPE can also call the CPF via upcalls provided during the initialization. Finally, communication between CPEs is done via MTI dispatch tables exchanged between CPF at binding.

5.4.2 Activation and cleanup mechanisms

The CPFs and CPEs are located in the same address space. The CPF loads the CPE control interface into the system by using standard dynamic library load mechanisms, and initializes it by calling the export `startup()` procedure.

The loading of an engine is usually triggered by the first instantiation of one of its front-end. When another CPF requires the loading of an engine, the engine is not loaded again, but a reference count is increased. Once the engine reference count reaches zero, the engine must prepare itself to be unloaded from memory. In particular, it must finish transmitting any unsent data and release all resource held. It must be left in a state where it can be immediately re-initialized by a call to `startup()`.

The `startup()` procedure must be called at least once for each CPF, and may be called multiple times by the engine loading mechanism. A matching `cleanup()` must be called for each successful `startup()` call. The CPE should maintain a reference count on a per-process basis.

As part of the initialization process, the CPF retrieves the CPE qECI and MTI dispatch table. From the dispatch table, the CPF is able to obtain entry points to the rest of the engine's functions.

Using dispatch tables (for both the MTI and the qECI) serves two purposes. First, it is more convenient for the CPF since a single call can be made to discover the entire set of entry points. Secondly, this enables the layered services to be formed and to operate more efficiently.

Two additional exported procedures may be available from engine implemented in C++. These are the `create()` and `destroy()` procedures. They allow to create and destroy objects from CPF. When an object is created, the procedure returns the object reference pointer. This pointer should be passed back to destroy the object when the object is no longer needed.

```
#define FUNC_DECL dll_export

typedef void * (*PFSTARTUP)    (void * param);
typedef void * (*PFCLEANUP)    (void * CPERef);
typedef void * (*PFCREATE)     (void * param);
typedef void * (*PFDESTROY)    (void * objectRef);

void * FUNC_DECL startup (void * param);
void * FUNC_DECL cleanup (void * CPERef);
void * FUNC_DECL create  (void * param);
void * FUNC_DECL destroy (void * objectRef);
```

5.4.3 Mapping between CPF API and CPE qECI

The CPF API and CPE qECI are similar in that all the basic functionality of the engine is provided by both interfaces. For example, for the transport protocol layer, TP, TPfrontend->Establish() maps to TPengine->establish(), similarly for release(), setQOS(), etc. The front-end converts the CORBA parameters provided at the front-end to platform specific parameters using the definitions provided in the qECI header.

Other calls such as AttachPort() and DetachPort() potentially map to a sequence of engine procedure call. Some front-end support procedures like getMTI(), getCID() are implemented by the front-ends and are not passed down to the engine.

5.4.4 Interface Extension Mechanism

Since the CPF API and CPE qECI are standardized, it is difficult for each individual engine to offer extended functionality by just adding entry points in the qECI dispatch table. To overcome this limitation, the MTI and qECI offers an i/o control procedure, `ioctl()`. This procedure accommodates engine developers who wish to offer engine-specific functionality extensions. This mechanism presupposes that an application is aware of a particular extension and understands both the semantics and syntax involved.

5.4.5 CPE qECI and MTI

The CPE qECI are interfaces to platform dependent transport component located in the datapath of the communication. They are defined in the C programming language so they can be easily be ported across various computing platform. The interfaces are very similar in flavor to the CPF control interface as the CPF is simply a CORBA front-end to the platform dependent engine.

mtistruct defines the data structures and mediatransfer interface common to all the CPEs. The media transfer interface is provided as a dispatch table. When two CPEs are bounded, the protocol stack component at the higher layer gets the MTI pointer of the lower layer protocol stack component via the VirtualPort MTIPtr readonly attribute.

Constants and data structures

mtistruct defines several return values for the media transfer operations ({send} and {recv} and their scattered i/o version) and various constants to be used with the {ioctl} operation. For simplicity, the prefix MTI_ has been omitted in the description below.

- [return codes] The return codes defined in mtistruct are for the media transfer operations ({send} and {recv} and their scattered i/o version). For simplicity, the prefix {MTI_} has been omitted.

  -[SUCCESS] Indicates the successful completion of the call.

  - [FAILURE] Indicates that the call did not succeed for a reason that has no return code value associated.

  - [NO_DATA] Indicates that no data was available. This is a typical return value for a {recv} call in non-blocking mode when non data is available.

  - [ALL_DATA_SEND] Indicates that all data was send to the next layer of the protocol stack.

  - [ALL_DATA_QUEUED] Indicate that the data was queued in the engine.

  - [BUFF_INSUFFICIENT] Indicates that the buffer provided by the caller is too small to receive the PDU.

- [CLOSED] Indicates that the port was closed before the call and that the operation cannot be performed.
- [NOTSUPPORTED] Indicates that the operation is not supported by the engine.

• [ioctl codes] The ioctl control codes define in mtistruct are to be used with the {ioctl} operation. The prefix {MTIIOCTL_} has been removed from the description below.

-[GETPDU] Returns the current maximum PDU sized allowed at to be transfer at the MTI.

-[GETMTU] Returns the maximum transmission unit of the network interface card. The engine should perform fragmentation and reassembly for PDU sent that have size larger than the MTU.

-[NONBLOCK] When a non-zero parameter is passed as third argument to the call, it sets the interface in non-blocking mode.

-[DATAREADY] Returns the number of bytes in the head of line packet in the queue. A null value indicates that no data is ready to be received. A negative value indicates a failure of the call.

-[FLUSH] Instruct the engine the flush its data buffer for the connection.

-[EVENT] Register a callback function with the engine. The third parameter to the {ioctl} call should be the event type and the fourth parameter a pointer to the callback function cast as void *.

• [callback events] The callback events are to be used jointly with the {ioctl} MTIIO-CTL_EVENT. When an callback is register with a specific event, the callback function provided is invoked at the occurrence of the event. The prototype of the callback functions for each event are provided in mti.

- [CLOSED] Register a callback function to notify that a connection has been closed. The opaque handle of the connection is provided as parameter to the callback.
- [ERROR] Register a callback function to notify that an error has occurred on the port. The opaque handle and error code are provided as parameters.
- [DATAREADY] Register a callback for the notification that data is available in the received queue.
- [ASYNCRECV] Register an asynchronous received function for the connection.

```
/* return values */

#define MTI_SUCCESS              0
#define MTI_FAILURE             (-1)
#define MTI_NO_DATA             (-2)
#define MTI_ALL_DATA_SEND       (-3)
#define MTI_ALL_DATA_QUEUED     (-4)
#define MTI_BUFF_INSUFFICIENT   (-5)
#define MTI_CLOSED              (-6)
#define MTI_NOTSUPPORTED        (-7)
```

```
/* ioctl values */

#define MTIIOCTL_GETMTU         1
#define MTIIOCTL_GETPDU         2
#define MTIIOCTL_NONBLOCK       3
#define MTIIOCTL_DATAREADY      4
#define MTIIOCTL_FLUSH          5
#define MTIIOCTL_EVENT          6

/* callback events */

#define MTIEVENT_ERROR          1
#define MTIEVENT_CLOSED         2
#define MTIEVENT_DATA_READY     3
#define MTIEVENT_ASYNC_RECV     4

/* structure for scattered i/o */

typedef struct _MTI_IOV_r {
   char * buf;
   long   len;
} MTI_IOV_r, *pMTI_IOV_r;
```

Methods

The methods of the media transfer interface are similar to the conventional one provided with BSD sockets, with an additional attribute parameter. This parameter can be used to customize the interface by specifying additional information about content of stream. For example, it can be used as a priority mechanism or to inform the CPE of the nature of the data (e.g., for an MPEG-2 adaptive enhancement layer). The return value of all media transfer operations is the number of bytes transferred or a negative number indicating that an error occurred during the transfer.

- [send] This methods send the data pointed by the parameter {buf} of length {len}. The {cid} is the opaque handle that can be obtained from the CPF attribute {CidPtr}.
- [recv] This method retrieve data and puts up to {len} bytes of data it in the buffer provided via the {buf} argument. If more data was received, it returns the {MTI_BUFF_INSUFFICIENT} error code.
- [sendiov] Scattered i/o version of the {send} method.
- [recviov] Scattered i/o version of the {recv} method.
- [ioctl] This method is used by the attached engine to setup some of the media transfer interface parameters. For example it can be used to flush the buffer queue. Please refer to the paragraphs on ioctl codes and callback events for a description of the define capabilities.

```
/* prototype for MTI callback functions */
typedef long (*PFMTICBERROR)      (short cid, short err,
                                   const char * message);
typedef long (*PFMTICBCLOSED)     (short cid);
typedef long (*PFMTICBDATAREADY)  (short cid, long size);
typedef long (*PFMTICBASYNCRECV)  (short cid, long size,
                                   const char * buf);
/* MTI operation definitions */

typedef long (*PFMTISEND)         (short cid, const char * buf,
                                   long size, short attr);
typedef long (*PFMTIRECV)         (short cid, char * buf,
                                   long * size, short attr);
typedef long (*PFMTISENDIOV)      (short cid, const pMTI_IOV iov,
                                   short iovlen, short attr);
typedef long (*PFMTIRECVIOV)      (short cid,  pMTI_IOV iov,
                                   short iovlen, short attr);
typedef long (*PFMTIIOCTL)        (short cid, long ctl_code, ... );

/* Interface definition */

typedef struct _MTI_r {
    PFMTISEND    send;
    PFMTIRECV    recv;
    PFMTISENDIOV sendiov;
    PFMTIRECVIOV recviov;
    PFMTIIOCTL   ioctl;
} MTI_r, *pMTI_r;

\input{arch/qos}
\input{arch/NetPproctable}  \clearpage
\input{arch/TPproctable}    \clearpage
\input{arch/TMuxPproctable} \clearpage
```

6 APIs for the Virtual Workshop

The Virtual Workshop is a distributed telelecturing application that allows multiple remote participants to collaborate in the style of an open workshop. Two modes of operation are supported in the application. In the first mode, there is a single sender and one or more receivers. The structure of the session in this case, takes the form of a multicast tree. In the second mode, there is only a single sender and receiver. The latter mode is useful in situations where only a point-to-point connection is needed. In the context of the workshop, the point-to-point mode is employed to implement a simple Video-On-Demand (VOD) feature where individual users receive a dedicated stream from a VOD server. In both modes, only uni-directional streams are supported.

6.1 Architecture

The architecture of the virtual workshop consists of 3 logically distinct classes of the following components:

- Virtual Workshop Graphical User Interface (VWGUI)
- Telelecture Builder (TLB)
- Session Directory (SD)

The VWGUI component implements a user interface that allows a human user to interact with the system. The VWGUI component implements no application logic but instead relies on the TLB backend for creation, configuration and management of the workshop sessions. The SD component implements a general purpose directory service for clients to lookup active sessions. Of the 3 components, only the VWGUI is located at the end-user system. The SD and TLB can be placed anywhere in the network but the location of the SD should be made known to the client. Figure 34 illustrates.

The thick lines indicate the direction of media flow (in this case a multicast session is shown). The dotted lines indicate control operations invoked by the TLB on the xbind daemon and xbind networked device.

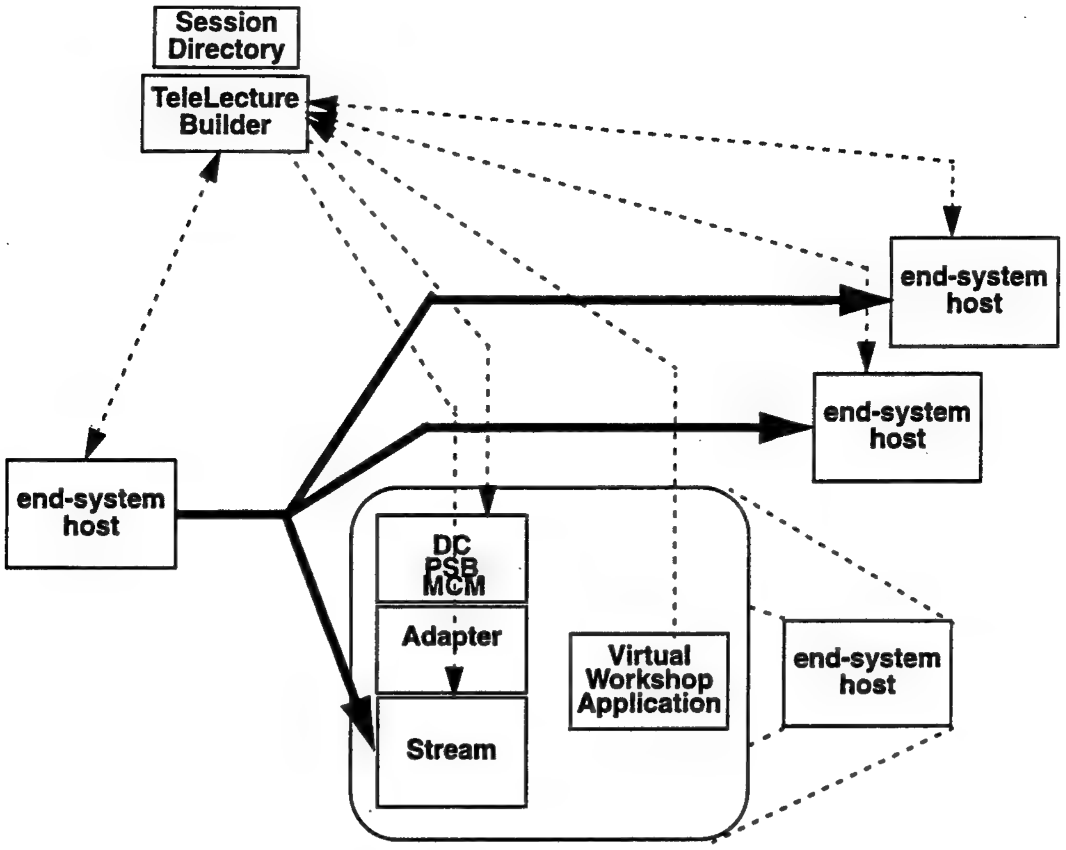


Figure 34 Architecture of the Virtual Workshop

6.2 The Virtual Workshop GUI (VWGUI)

The VWGUI is a modular CORBA client implemented in the Java language. The GUI supports the following functionality:

- Session lookup and browsing through a well-known SD.
- Session creation, joining and leaving.
- Authoring of simple tree-structured VOD programs.

- Viewing of pre-stored VOD programs.
- Simple stream device control.

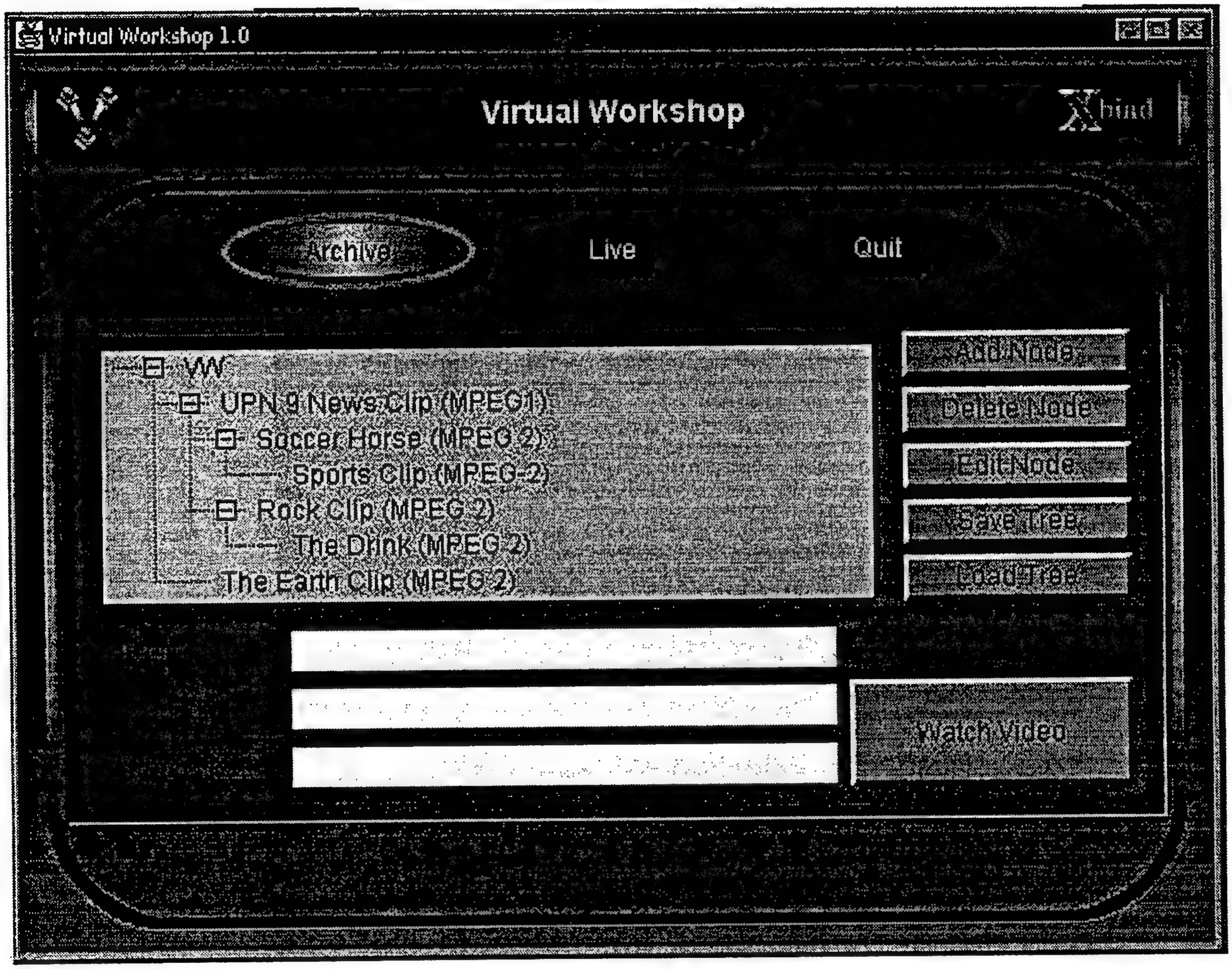


Figure 35 Virtual Workshop Graphical User Interface

6.3 The Telelecture Builder (TLB) and Session Directory (SD)

The TLB is a high level service that constructs telelecture sessions from distributed xbind BIB components. In the current implementation, the TLB incorporates the SD and is implemented as a CORBA server in the Java language. The programming interface of the TLB allows a client to create and manage multiple multicast and unicast sessions and issue simple stream commands to connected devices. The interface of the SD allow clients to query available sessions and the appropriate information associated with them. The user interface of the TLB is presented in Figure 36.

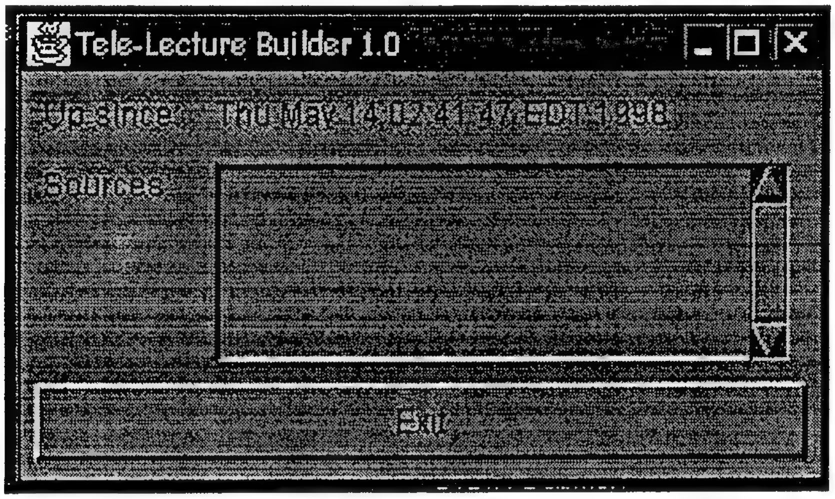


Figure 36 User Interface of the Telelecture Builder

Conceptually, the TLB can be viewed as a logical graph building algorithm. The graph constructed represent the distributed state information of a session. This includes information about the sending device, receivers, endpoint addresses, stream format and Quality of Service (QOS) parameters. The TLB relies on the following APIs for its operations:

- Device Controller --DC (in xbind daemon)
- Protocol Stack Builder --PSB (in xbind daemon) --sometimes called TC, Transport Controller.
- Multicast Connection Manager --MCM (in xbind daemon)
- Stream (in xbind networked device)

The TLB executes the following service creation algorithm upon receiving a request to create a session:

1. It checks the sender end-system to verify the availability of an appropriate device to handle the user specified format.
2. It instructs DC on the sender host to create the appropriate `Stream` object.
3. It instructs the MCM to generate a multicast tree id.

When it receives a request to join a receiver to an existing session, it executes the following algorithm:

1. It checks the receiver end-system to verify the availability of an appropriate device to handle the media stream format used in the session.
2. If it is the first receiver, it instructs the sender host's PSB to construct the appropriate transport protocol stacks based on the QOS requirements of the media stream.
3. It then attaches the `VirtualTP` object to the `Stream` object on the sender side and signals the device to begin transmission.
4. It instructs the receiver host's PSB to construct the appropriate `Stream` object, protocol stacks, and attaches the former to the latter.
5. It signals to the receiver's device to begin decoding the rendering the incoming stream.

```
#include "Stream.idl"
#include "VirtualNetP.idl"
#include "EndPoint.idl"
#include "QOSTP.idl"
#include "VirtualPort.idl"
#include "TC.idl"
#include "DC.idl"

typedef long McastTreeId;
typedef long SessionId;
typedef long BranchId;
typedef sequence<SessionId> SessionIdList;
enum TeleLectureUser { SENDER, RECEIVER };

struct ReceiverInfo {
   string dst_host;
   BranchId branch_id;
   VirtualNetP dst_vnetp;
   VirtualPort dst_vport;
   Stream dst_device;
   string dst_device_name;
   TC dst_tc;
   DeviceController dst_dc;
   EndPoint dst_endpoint;
```

```
    long join_time;
};

typedef sequence<ReceiverInfo> ReceiverInfoList;

struct LectureSessionInfo {
    SessionId session_id;
    McastTreeId tree_id;
    string description;
    string URL;
    string src_host;
    string src_device_name;
    string format_name;
    TransportQOSSpec tp_qos_spec;
    VirtualNetP src_vnetp;
    VirtualPort src_vport;
    Stream src_device;
    TC src_tc;
    DeviceController src_dc;
    EndPoint src_endpoint;
    long num_participants;
    ReceiverInfoList receiver_info_list;
    long creation_time;
};
typedef sequence<LectureSessionInfo>LectureSessionInfoList;

interface TeleLectureBuilder
{
  void pause(in SessionId s,in BranchId b,
                        in TeleLectureUser user_type) raises (Reject);

  void resume(in SessionId s,in BranchId b,
                        in TeleLectureUser user_type) raises (Reject);

  SessionId createSession(in string src_host,
                        in string format_name,
                        in string description,
                        in string URL,
                        in string src_dev_subtype,
                        in StreamFormat src_config)   raises (Reject);

  SessionIdList querySessionIds();

  LectureSessionInfoList QuerySessions();

  void destroySession(in SessionId session_id)       raises (Reject);

  LectureSessionInfo getSessionInfo(in SessionId session_id)
                        raises (Reject);

  BranchId joinSession(in string username,in string userkey,
                        in SessionId session_id,
                        in string dst_host,
                        in string dst_dev_subtype,
                        in StreamFormat dst_config)   raises (Reject);

  void leaveSession(in SessionId session_id,
                        in BranchId branch_id)       raises (Reject);
};
```

Structures

ReceiverInfo struct.

- dst_host - the hostname of the receiver
- branch_id - a unique number identifying the receiver from others in the multicast session
- dst_vnetp - the object reference of the receiver's VirtualNetP instance
- dst_vport - the object reference of the receiver's VirtualTP instance
- dst_device - the object reference of the receiver's Stream instance (representing the playback device)
- dst_device_name - the type of rendering device used by the receiver to playback the multimedia stream
- dst_tc - the object reference of the receiver's PSB instance
- dst_dc - the object reference of the receiver's DC instance
- dst_endpoint - the endpoint terminating this branch
- join_time - the time the receiver joined the multicast session

LectureSessionInfo struct.

- session_id - a unique number identifying this session on the TLB
- tree_id - the id of the multicast tree as assigned by the multicast connection manager
- description - a textual description of the session
- URL - an optional URL describing the session
- src_host - the hostname of the source
- src_device_name - the type of source device used to generate the media stream
- format_name - the format of the media stream
- tp_qos_spec - the transport protocol layer QOS parameters associated with this session
- src_vnetp - the object reference to the VirtualNetP instance at the source
- src_vport - the object reference to the VirtualTP instance at the source
- src_device - the object reference to the Stream instance (representing the source/coding device)
- src_tc - the object reference to the PSB at the source
- src_dc - the object reference to the DeviceController at the source
- src_endpoint - the endpoint at the source
- num_participants - number of receivers in the session

- receiver_info_list - a sequence of records containing detailed information about each receiver
- creation_time - the time when the session was created

Methods

pause. This method issues a pause request to device of the receiver on branch `b` of session `s`.

resume. This method issues a resume request to the previously paused device of the receiver on branch `b` of session `s`.

createSession. This method creates a new session with `src_host` as the source. The media stream carried on this session is of the format `format_name`. The session is described in `description` and further information of the session may be found at `URL`. The source device is to be instantiated with subtype `src_dev_subtype` and passed the parameter `src_config` during creation.

querySessionIds. This method returns a list of session ids created.

QuerySessions. This method returns a sequence of descriptor describing each session in full details.

destroySession. This method destroys an existing session identified by `session_id`. All resources at the sender and all receivers are released.

getSessionInfo. This method returns detail session information of a session identified by `session_id`.

joinSession. This method joins a receiver identified with `username` and `userkey` to an existing session identified by `session_id`. The receiver's host is given by `dest_host` and subtype and parameters to be used to instantiate the receivers device is given by `dst_dev_subtype` and `dst_config` respectively.

leaveSession. This method removes a receiver identified by `branch_id` from the session identified by `session_id`.

7 APIs for Service Management

We are witnessing an explosive growth in distributed multimedia applications and services, such as video conferencing, video-on-demand and collaborative distributed environments, together with an increasing variety of underlying network technologies. As a result, telecom networks and environments are becoming more and more complex.

This situation creates major challenges for managing telecom multimedia environments. One of these challenges relates to the variety of services and technologies. Management architectures have been developed to address this problem. The TMN framework [ITU92], for instance, introduces an architecture that defines layers of increasing levels of abstractions (element management layer, network management layer, service management layer and business management layer). The service management layer deals with the services provided by the telecom system, including the multimedia services.

A second challenge is to design the new services (and technologies) in such a way that they can be managed on the appropriate level of abstraction. The fact is that most management systems today are built as an afterthought to service delivery systems. This makes it difficult to add the hooks for monitoring and controlling service activities later. More importantly, the service delivery system might have been designed and/or implemented differently, had the management requirements been taken into account during the analysis phase of the system's development.

This chapter addresses the problem of designing and realizing manageable multimedia services. Our approach centers around defining a set of cooperating objects involved in the interaction between the service delivery and the service management system. The model we propose describes the interface between the two systems, and covers--in a generic way--the aspects of instantiation of a *service session*, its access by a user and its management by the operator. (A service session represents the information handled by all processes involved in providing a service; a session has a start, a duration and an end.) Our model suggests a generic solution to a recurring problem in service design. The model needs to be customized for a particular service and refined according to management requirements and available resources. In this sense, we are proposing a *design pattern* [GAM95] for making telecom services manageable.

This work focuses on service management in the sense of supervising the service delivery system. The management task includes monitoring and controlling the service delivery system to ensure that it operates as intended, maintaining the service-level agreements while achieving management objectives. The task is performed on a slower time-scale than the functions executing in the service delivery system. This activity relates to the OSI functional areas of performance and configuration management, and to the functions in the service management layer of the TMN architecture.

In the domain of service management we are considering, two activities can be identified: a) *managing the set of controllers* (i.e., routers, connection managers, etc.) which comprise the functionality of the service delivery system, and b) *managing the set of service instances* (such as audio multicast connections or VOD sessions) which are dynamically created, modified and terminated during the operation of the service delivery system. While we have presented some results on activity a) in [CHA96b], this paper focuses primarily on activity b), namely, on the management of service instances (or sessions) and aggregations of such instances.

The rest of the chapter is organized as follows. Section 7.1 outlines the generic object model we propose for making services manageable. In Section 7.2 the multicast service addressed in Chapter 4 is made manageable based on our model. A first implementation of the system on the two platforms mentioned above is described in Section 7.3. Finally we introduce Peer to Peer management in the last section.

Related Work

The need for making services manageable has been widely recognized. Work on this topic is being pursued within the areas of distributed computing, internet networking and--of course--telecom systems. Within the field of distributed computing, Schade et al. [SCH96] propose an extension to CORBA IDL, which allows for specifying the management interface of service components in a distributed environment. The authors provide a library for implementing such a management interface. An approach to make services manageable for an ISP (Internet Service Provider) is described in [BHO97]. The authors address the problem of fault diagnosis in a multi-domain environment. Using the concept of service contracts, they suggest an architecture that provides an ISP with management (monitoring) interfaces to service components belonging to the domain of another ISP. In the telecom field, the activity most related to our work is the TINA initiative [TINA95]. TINA also applies an object-oriented methodology for developing both the service delivery and the service management systems. So far, the TINA effort has concentrated on the service delivery system; aspects of service management as presented in this paper have not been addressed until now. The EURESCOM P610 project has recently been launched to develop an architecture for multimedia service management [EUR97]. Because it addresses issues similar to those discussed in this paper, results from this project will potentially influence our research.

7.1 A Generic Object Model for management

This work focuses on service management in the sense of supervising the service delivery system. This activity relates to the OSI functional areas of performance and configuration management, and to the functions in the service management layer of the TMN architecture. The management task includes monitoring and controlling the service to ensure that it operates as intended, maintaining the service-level agreements while achieving management objectives. The task is performed on a slower time-scale than the functions executed in the service delivery system.

The service delivery system contains a set of interacting controllers that perform the tasks of service creation and service delivery in a distributed fashion [CHA96b]. Specifically, the system consists of a large number of controllers, with many instances of the same controller class spread throughout the system to support distributed operation and fault tolerance. A limited number of controller classes exist, each of which encapsulates a specific functionality, such as running a network control algorithm (admission control, routing, etc.). Most controllers are realized in software. They can be designed as communicating active objects. They are multi-threaded, i.e., they have separate address spaces, and they communicate by exchanging messages, which allows them to run on different time scales. Furthermore, they are designed as persistent objects, which are created during the initialization phase of the control system.

Specifically, in this domain of service management we are considering, two activities are identified: a) *managing the set of controllers* (i.e., routers, connection managers, etc.) that comprise the functionality of the service delivery system, and b) *managing the set of service instances* (such as audio multicast connections or VOD sessions) which are dynamically created, modified and terminated during the operation of the service delivery system. In the following we focus on the management of service sessions.

Our approach for making a service manageable introduces a new type of controller in the service control system: a dynamic object that is created for each new session. This object, which represents a service session (or service instance), makes itself accessible to the management system by registering with a management object, the service aggregator.

Figure 37 shows the object classes involved in the service management activity.

For each type of service there is a service factory which handles the requests for new service sessions and coordinates the service creation/instantiation process. For example, a service factory for unicast VCs receives a request for a new VC and contacts the necessary controllers (connection managers and routers) in order to set up the new VC. As a result the user who requested the service will receive a handle to use the new VC. The service factory object abstracts the specific scheme used for the service instantiation process.

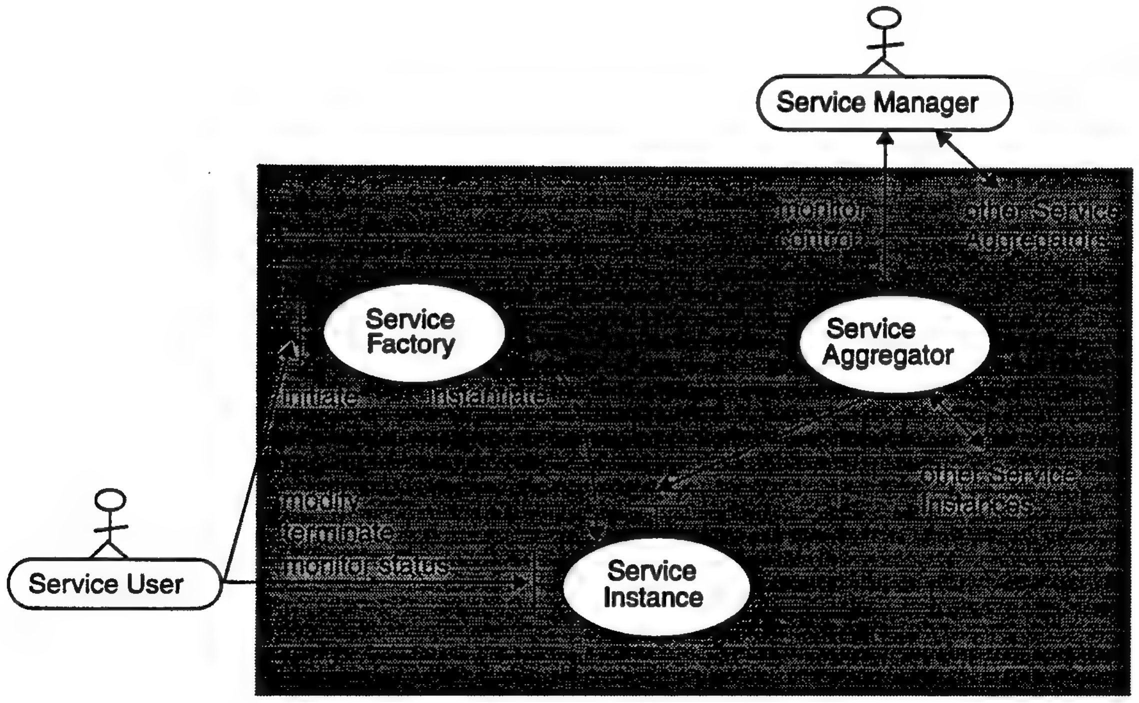


Figure 37 Cooperating objects enabling service management.

The service instance represents the status and capabilities of an existing service session. Figure 37 shows the service instance with two types of interface. One is accessed by the service user and represents the control capabilities (status, modify, terminate) a user has once the service session is created; the other is accessed by the management system and represents the management capabilities.

The service aggregator has two purposes. First, it provides a single point of access for the service manager to monitor and control the existing service sessions of a particular service type. Second, it provides aggregated or abstracted views of the service instances and allows the manager to manipulate sets of service instances. The service factory and the service aggregator are *persistent* objects. They are created and initiated at service deployment time and generally stay alive as long as the service is provided. The service instance, on the other hand, is a *dynamic* object. It is created when a service session is instantiated by the service delivery system, and it is deleted when the session is terminated.

The model we are proposing is generic in the sense that it describes object classes, their relationships and their interactions for the purpose of making service sessions manageable. In fact the set of operations at the interfaces, both offered to the user and to the manager, are generic for any kind of service.The specific functionality of, say, a service factory, depends on the particular service. Similarly, the lifetime of service instances differ from service to service in a typical range from minutes to days. Service instances of different types can be related via service composition. For example, a teleconference may be built using a set of unicast and multicast VCs.

An important aspect of our model is that the designer of a service has several degrees of freedom when developing service management functionality. The choices include the selection of the state information and functionality of the service instances, the amount of information kept in the service aggregators and the consistency requirements for the management data in the aggregators. All these factors influence the *cost* of managing the service in terms of design complexity during the development phase and in terms of processing and computational resources needed during the operational phase.

7.2 Managing Connection Management

mcast is an ATM multicast service based on an object oriented design. It provides the capability of transporting multimedia information from a source to one--or more than one--destination. An *mcast* session belongs to one specific service class, i.e., video, data or voice. (A service class defines the performance characteristics and the performance requirements of a cell flow.) A destination can request to join or leave an *mcast* session. The termination of a session is requested explicitly by the source.

The *mcast* design includes three classes of controller objects: the *mcast* directory service (MDS), the *mcast* router (MR) and the *mcast* connection manager (MCM). The first two classes are global controllers, i.e., they maintain global knowledge about the service. In contrast, the MCM keeps local knowledge (from a host or switch).

The main purpose of MDS is to allow prospective destinations to get information about the currently existing *mcast* service sessions. MDS also keeps partial information about the current graph that represents each *mcast* session for routing purposes. MR computes the route to a prospective destination upon request. It first selects an intermediary node in the tree to act as source in the route. The NodeServer encapsulates the switch capabilities to establish and terminate local unicast connections. It realizes the interface between the service delivery system and the network resources.

There is one MCM instance for each node (host or switch). The operations offered by MCM are associated with two interfaces: one offered to the service user and the other offered to a peer MCM. The operations at the service user interface are: `initiate`, `join`, `renegotiate`, `leave` and `terminate`. They all refer to one specific *mcast* session. An MCM receiving a `join` request coordinates the reservation of resources by interacting with the peer MCMs along the route to the new destination. Upon a `leave` or a `terminate` request, the interaction among MCMs is hop by hop. This interaction can happen in either direction--from destination to source or vice versa --depending on which controller initiates the process: the source, a destination or a third-party.

All interactions among controllers are asynchronous and are based on a reliable datagram service with FIFO property. The *mcast* service design addresses the issue of consistency of multicast trees under concurrent operations. This aspect was addressed in Chapter 4.

Using the model introduced in the previous section we present a design for managing service instances and aggregations of *mcast* sessions. The management system monitors the set of existing *mcast* sessions and is able to control the system by creating new *mcast* instances, modifying the state of existing instances and terminating instances.

Making a system (a service) manageable means making both the system that is being managed and the management system "aware" of each other, by providing addresses at which to communicate. In our model in order to make a service manageable, (1) the service delivery system has to maintain the service instances and (2) the service management system has to be able to access them.

To accomplish (1), we need to identify the controller(s) offering the service interface to the user. In our service management model, this interface is provided by the service factory object (creation) and the service instance object (all other operations). In *mcast,* this interface is provided by the *mcast* connection manager (MCM). MCM provides the following operations to the user: `initiate`, `join`, `renegotiate`, `leave` and `terminate`. Figure 38 shows an example in which the MCMs, acting as service factories, generate the service instance objects.

To accomplish (2), upon creation, these objects make themselves accessible to the management system by registering with the *mcast* service aggregator.

To refine the design, the following questions need to be answered: What is the state about an *mcast* session that is to be kept in a service instance object? What is the mechanism by which service instances are updated? What is the level of aggregation within the service aggregator? What is the interface between the service aggregator and the service manager? What do we want to visualize at the SM GUI?

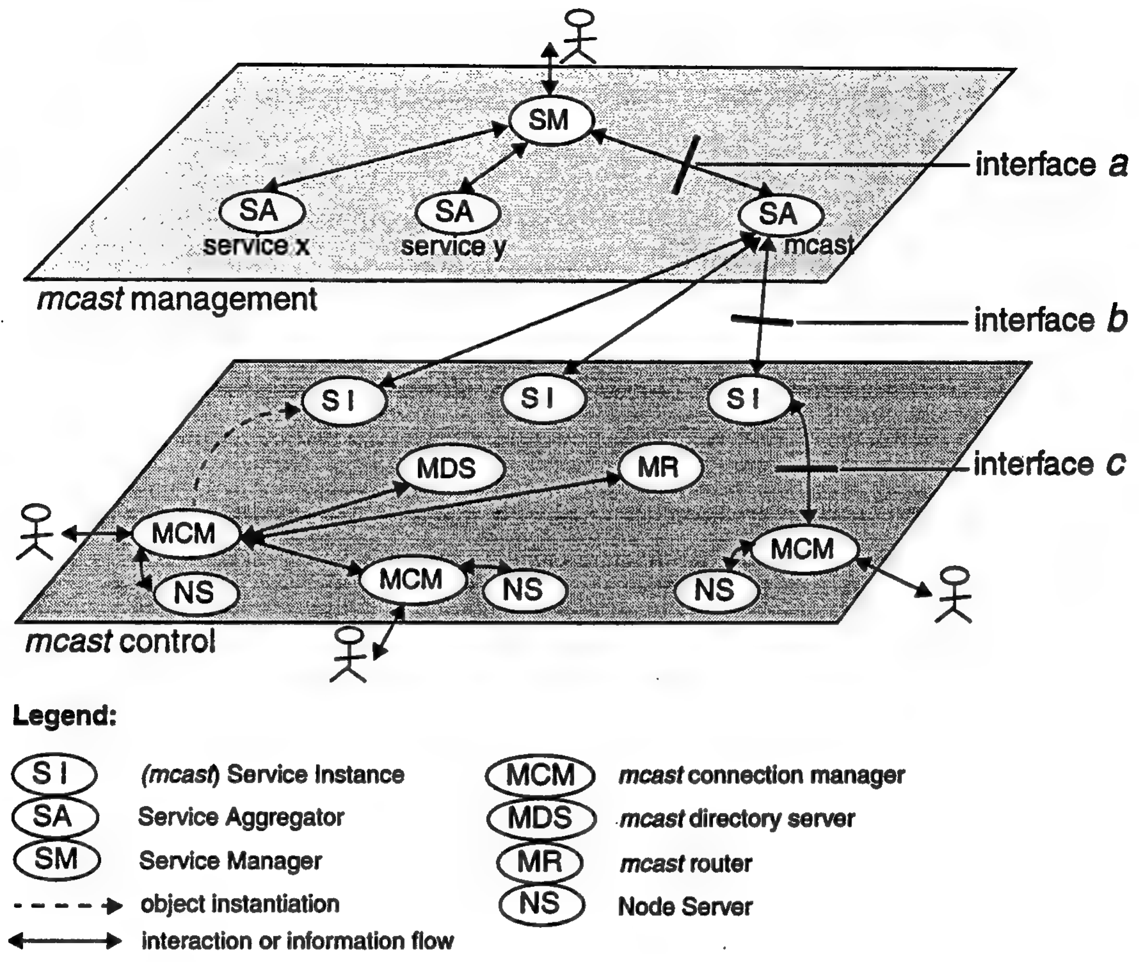


Figure 38 Management of *mcast* service sessions.

Next, the interfaces *a, b* and *c* shown in Figure 38 need to be defined. Each of them will affect the interfaces of two controllers.

7.2.0.1 Monitoring *mcast* service instances

For monitoring, either the management system polls the system being managed for its state, or the system being managed sends notifications about its state to the management system. The information passed in either way might be either the whole state or event based, with incremental information about the state. Designing an information system consists on deciding how to combine these four styles of passing information.

At the interface *a*, both polling (from service manager to service aggregator) and sending notifications (from the service aggregator to the service manager) are supported, for a single service instance and for a set of instances. The operations defined are:

- for a session: `newSession`, `modifiedSession`, `terminatedSession` as notifications and `getState` as a polling operation (this allows the service manager to obtain the current state of a session);
- for a set of sessions: `getState`, as polling operations. In the current implementation a set is specified by enumeration.

Monitoring in *b* and *c* is accomplished via notification messages. In *b* these messages, which travel from the service instances to the service aggregator, are: `newSession`, `modifiedSession` and `terminatedSession`. In *c* the only message defined is `modifiedSession`. Figure 39 summarizes a first design for monitoring.

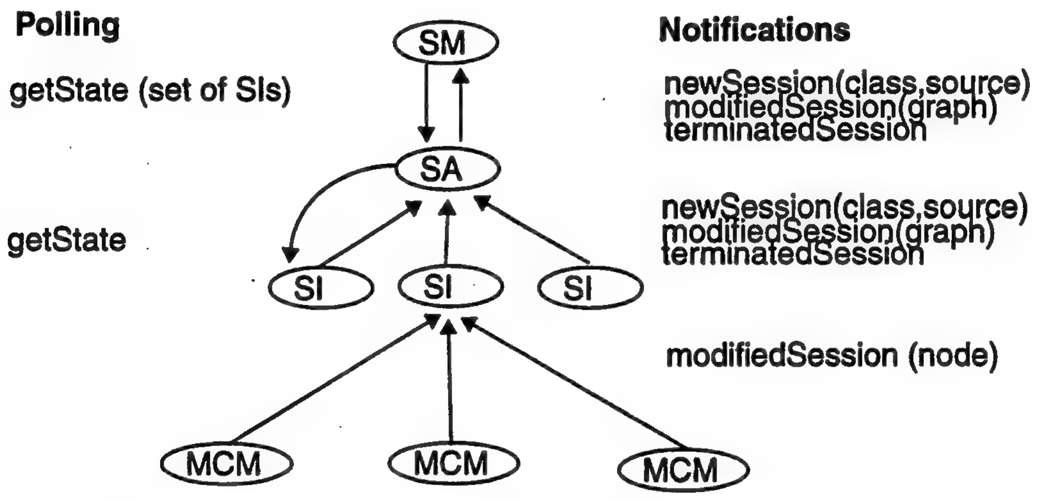


Figure 39 Cooperating objects enabling service management.

We define the following state for a service instance object:

- service class: resource requirements and expected cell-level QoS;
- connectivity graph: source, destinations and network nodes involved;
- measurements on call-level traffic: joining and leaving rates;
- measurements on the call-level QoS provided: rejected joins; time to establish a join;
- lifetime of the session.

MCMs at source instantiates and terminates the SIs. MCMs at destinations and intermediate nodes update SIs. This implies that all MCMs need to have a reference to the SIs as they are created. This could be feasible if the reference can be derived from the *mcast* sessionId. If not, MDS could store the SI interface for the prospective destinations and take care of intermediate node updates.

The Service Aggregator:

- keeps the number of sessions per service class, and has access to each SI;
- generates aggregated (abstracted) views, possibly per service class and per destination.

The getState request allows to access the whole state of one or a set of service instances. Different predicates could be provided.

Making the system manageable consists of telling MCMs to maintain SIs. We are looking into ways to control this maintenance system from the management interface.

For the GUI at SM, we are making use of a 3D GUI that provides the view of the network graph, and on top it shows the *mcast* sessions. Each session is represented as a colored sphere on top of the node-source. Color reflects the service class. Sessions can be "selected" by clicking on them (or all). Trees (connectivity graphs) can be visualized for selected SIs; these are shown on top of the network graph to reflect the routing.

7.2.0.2 Controlling *mcast* service instances

By controlling we mean affecting the state of the SIs. Our management system is able to perform the following operations: `create` a service instance, `modify` one service instance or a set of service instances, and `terminate` one service instance or a set of service instances.

For example, the management system connects a video broadcast to the network. This procedure includes the service manager requesting a new session via the *mcast* service aggregator, which, in turn, contacts an MCM acting as the service factory. As a result, the availability of the video broadcast is advertised through the *mcast* directory service.

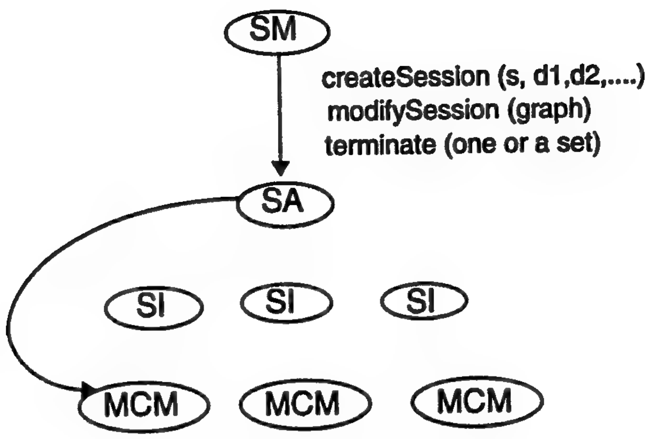


Figure 40 Controlling *mcast* instances.

Note that these management operations do make use of the service delivery system for its execution, as shown in Figure 40. The `create` operation issued by the manager identifies a source and a list of destinations: the SA will execute the `initiate` operation and as many `join` operations as necessary on the corresponding MCMs. The `terminate` operation specifies a set of instances: the SA will pass the request to the MCMs at sources.

The `modify` operation applies to the state of a service instance or a set of instances. Given the above definition for the state of a service instance (resource requirements, connectivity graph and traffic statistics), `modify` can apply to either the requirements or the graph. If the modification request applies to the service class, the request is translated by the delivery system into a `renegotiate` operation. If it applies to the graph, then it is translated into a `join` or `leave` operation.

7.2.0.3 The interface for the service manager

Figure 41 shows the 3D operator interface used in both platforms for the management of the *mcast* service. The network topology is shown at the lowest layer. Different colors on these bars represent different service classes, such as video, audio and data. The balls above represent *mcast* sessions for which the nodes below act as sources. For the six sessions starting at the node nearest the viewer (bottom left), the multicast trees are shown.

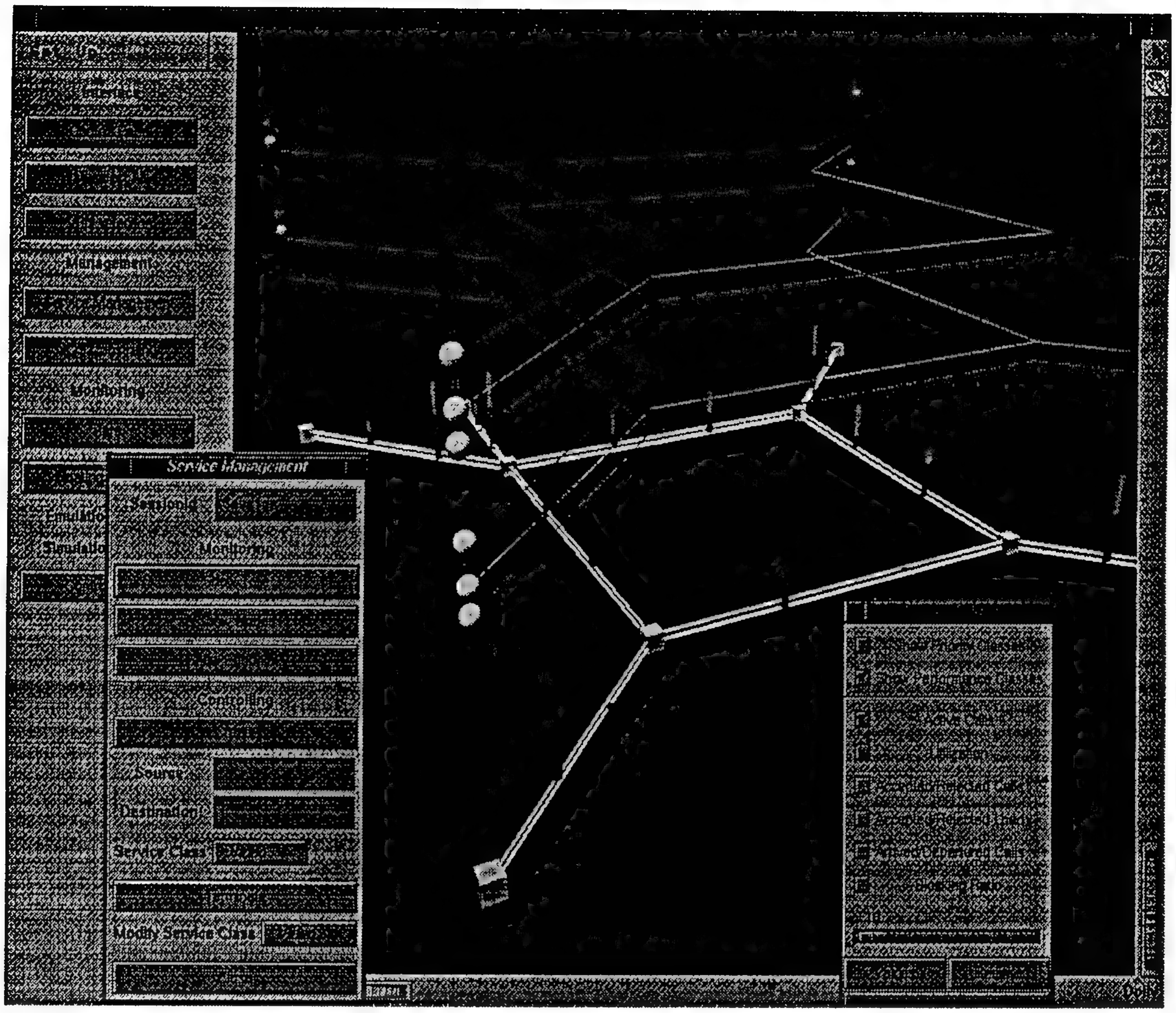


Figure 41 The operator interface displaying the current link load and the current sessions of the *mcast* service. The window in the lower left corner allows the operator to perform service management operations on selected *mcast* sessions.

7.3 Extending Management to other Services

Our work up to now has focused on the control subsystem (*mcast*). *mcast*, as described above, is a connectivity service that provides a connectivity graph between network endpoints. As such, it does not address (1) the communication between the multimedia devices which actually produce/consume the information, and the network endpoints; (2) the transport aspect of the multimedia flows. In order to directly support networked multimedia applications, *mcast* needs to be extended to an *Application Level Service* (ALS), in which end-devices and the transport of flows are considered. The Virtual Workshop service explained in Chapter 6 is an example of an ALS service.

Applying the model for managing service instances (Section 7.1) to both *mcast* and an ALS service can lead to two co-existing management systems. This is shown in Figure 42. Having several co-existing management systems may be necessary in some cases, for example, in a telecom environment in which *mcast* and ALS are offered by different providers. It may also be useful in a single-provider environment, if *mcast* is accessed by a variety of ALS.

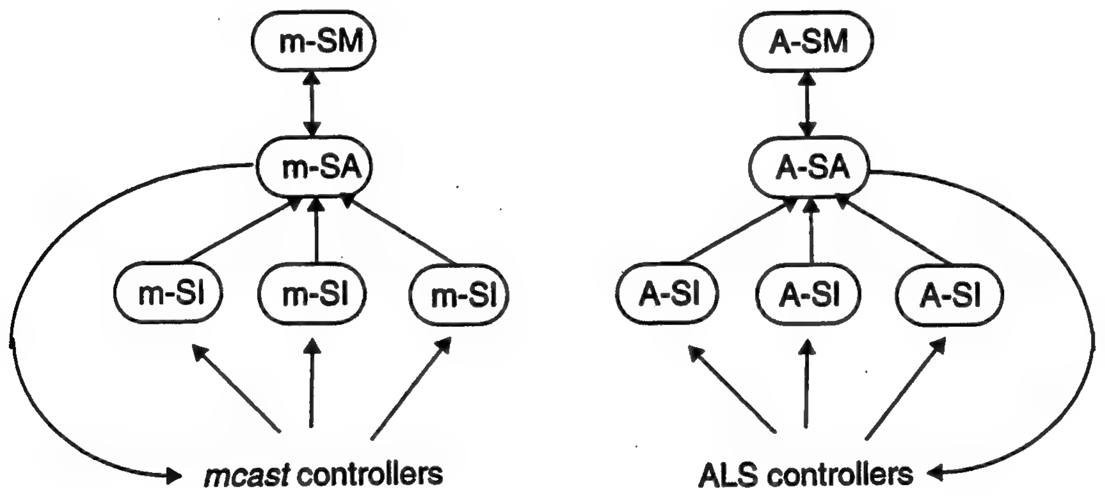


Figure 42 Applying the Service Management model to *mcast* and ALS.

The design outlined in Figure 42 raises the question of how to refine it such that the consistency requirements between *mcast* and ALS service instances (and consequently between *mcast*-SA and ALS-SA) are taken into account. Our current design is based on the assumption of a single-provider environment and includes an ALS that is a straightforward extension of *mcast*. In our case, there is a one-to-one relationship between *mcast* and ALS service instances. We therefore model an ALS service instance as a container object that includes a single *mcast* service instance. As a result, we have implemented only the ALS management system, leaving out the *mcast* SA and SM controllers.

The design for ALS management, as outlined above, is consistent with the approach to service management as expressed in TINA's Service Component Model, which is part of the TINA Service Architecture [TINA-SA]. By "consistent" we mean that there is a direct mapping between functional components of both models. Our model is simpler in the sense that it does not address issues of peer-to-peer management, subscription and accounting, and certain aspects of life-cycle management, such as service deployment and withdrawal. On the other hand, our model is more generic, since it can be applied to connectivity services, as demonstrated by the example of *mcast*. (TINA has developed it's own management model for that purpose, as part of the Network Resource Architecture [TINA-NRA].) Most important, however, our model is based on asynchronous interaction among components, which enables management and service control operations to be performed on different time-scales, and which facilitates the concurrency of operations within the control and management systems.

7.4 Implementation aspects

We have implemented the *mcast* service and the management capabilities described in previous Sections on two platforms, namely, a) on a high-performance emulation platform, which allows us to develop a software prototype and to study its dynamic behavior and scaling properties in various scenarios [CHA96a], and b) on a broadband testbed running xbind [LAZ96], a CORBA-based multimedia networking platform, on which services are fully implemented and the transport between multimedia devices is realized.

Our requirement is that the implementation design of the service control and management systems can be done in such a way that their code runs on both platforms. Also, the GUI enabling service management functionality by an operator must run on both platforms.

7.4.1 The emulation platform

Figure 43 shows the building blocks of the emulator platform. The emulated system and emulation support modules consist of a set of objects that communicate by exchanging messages, using functions provided by the simulation kernel. The emulated system module represents the prototype system under evaluation. In our case, this module contains the controllers of the *mcast* service delivery and management systems. Generally, each controller is implemented as a C++ object.

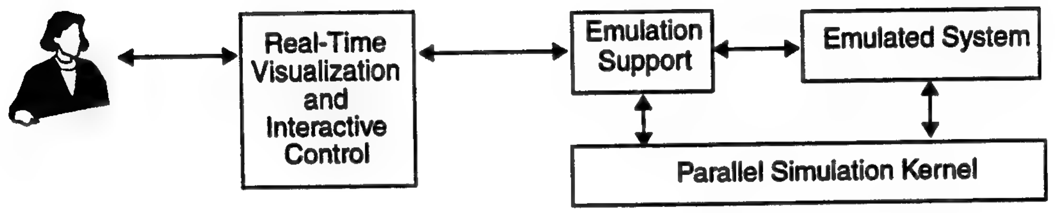

Real-Time Visualization and Interactive Control
Emulation Support
Emulated System
Parallel Simulation Kernel

Figure 43 Building blocks of the interactive emulation platform.

The *simulation kernel* controls the execution of these objects and ensures that messages are processed in the correct order. It is realized as a parallel discrete event simulation (PDES) system, using a window-based synchronization protocol. In order to support real-time visualization and interactive control of the emulated system, the kernel controls the progression of the simulation time, constraining it by the progression of the processor time. The module for real-time visualization and interactive control contains a graphical interface which provides 3-D visual abstractions of the system state.

Both the emulated system and the simulation kernel (coded in C++) run on the same machine. The real-time visualization and interactive control module resides on an SGI Indigo2 workstation. It is written using Open Inventor, a 3D graphics tool kit based on Open GL. The emulation support module is distributed on the two machines.

7.4.2 The Management GUI

7.4.2.1 Architecture

The user interface for management makes use of two communication processes--*kcomm*--to interact with "external" systems. The user interface process is called *Net3D*. The communication between the kcomm processes and the interface Net3D goes through shared memory (Figure 44).

There are two shared memory blocks. One is for passing information from the kcomm process to Net3D (*shared_measure_t*), the other is for passing commands from Net3D to the kcomm process which sends the commands to the Telesoft object *MonCtrlManager (shared_cntl_t)*.

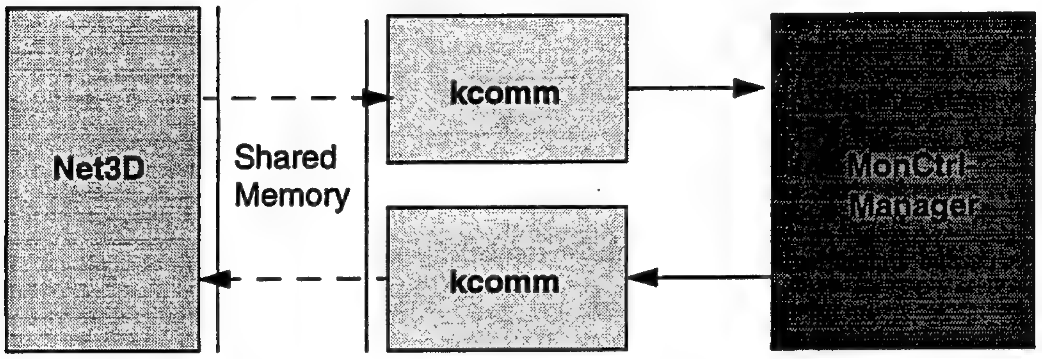


Figure 44

Objects that inherit from *MonCtrlObject* can be monitored. In our work the external system the user interface needs to communicate with consists of one object: the *mcast* Service Aggregator. Therefore the *mcast* Service Aggregator (SA) will inherit from MonCtrlObject.

For each object that is monitored, a slot is reserved in the shared_measure_t shared memory. Classes of the interface that want to receive monitoring information from certain objects (for example, from SA) must have the class *Stat* as a base class. The class Stat is an abstract base class. The *update()* method of a Stat object is called repeatedly (in our implementation with a cycle time of 2 seconds). This method should analyze the latest monitoring sample and update its visualization.

7.4.2.2 Visualization

There is one instance of the *Network* class. Its method *configure()* is called once at the beginning. That method parses the topology file (which is a command line parameter to *Net3D*) and creates objects of *NetNode* and *NetLink*. Each NetNode and NetLink object creates the necessary *OpenInventor* objects to be displayed (in the method *draw()*).

When the *Service Management* button is selected, an instance of the class *ServiceManagement* is created. It also displays a window. That instance immediately creates an object of the class *MCstat*. This class receives monitoring information from the *mcast* Service Aggregator object. First, only bubbles are displayed that represent one *mcast* session each. Each bubble is an object of the class *MCsession*. MCstat contains a list of MCsession's. If, for a certain bubble (or *mcast* session), the tree should be displayed, then an object of the class *MCfullSession* is created, which displays the tree and contains the knowledge of the whole tree.

The update() method of MCstat is more complicated than the one of other Stat objects. The reason for this is the following: other Stat objects receive monitoring information that reflects the current state of a certain object. Therefore even if the update() function is called twice for the same monitoring information, or if the update() function is not called at all one time, it does not matter. On the other hand the *mcast* SA sends messages (as monitoring updates) to MCstat that reflect changes in its state, but it does not send its whole state. Therefore it is crucial that no message is lost and that no message is handled twice. For that reason a simple protocol has been introduced between the *mcast* SA and the MCstat object.

A potential problem in the update() function is that no message may take up more space than a certain amount (e.g. 2048 Bytes). If it does, the interface will crash.

7.4.3 APIs for the management system

Most of the operations described in Figures 39 and 40 have been implemented.

7.4.3.1 Multicast Service Instance

A multicast service instance is an object dynamically created by an MCM. It keeps a list of links and a list of nodes that take part on the tree. Both lists just contain id's which are *typedef*'s of *short*:

```
List<MCNodeId> d_nodes;
List<LinkId> d_links;
```

7.4.3.2 Multicast Aggregator

The multicast aggregator keeps for each *mcast* session the following information:

```
class MCServiceInstanceData {
 ServiceClass serviceClass;
 McastId session;
 List<MCNodeId> nodes;
 List<LinkId> links;
 int stateChanged;
};

List<MCServiceInstanceData*> d_serviceInstances;
```

The multicast aggregator notifies the service manager (aka, user interface, manager station) about new and terminated sessions. It also informs the manager when the tree of certain multicast sessions changes. The manager can decide, by using the user interface, which multicast sessions should be monitored. A list of these sessions is kept by the multicast aggregator:

```
List<MCServiceInstanceData*> d_monitored;
```

The method getState() suggested in Figure 39 is not provided in the current implementation. Instead messages from SA are sent to SM every 2 seconds with the whole graph for each selected instance, if and only if there were any changes in the last 2 seconds. The method modify suggested in Figure 40 has not been implemented.

7.4.4 The broadband testbed

We use CORBA technology [OMG96] for building the *mcast* service delivery system as well as the management system on the broadband testbed. CORBA offers a flexible object oriented environment for the design of distributed software. The CORBA implementation we use is based on OrbixV2.0 [IONA96].

The broadband testbed is a local ATM network that connects a set of workstations and multimedia devices, such as cameras and microphones. The testbed runs xbind [LAZ96], a CORBA-based open platform for service creation and implementation. xbind defines and implements a collection of CORBA objects called the *Binding Interface Base* (BIB). The BIB provides access to abstractions of networking resources and multimedia devices. For example, the VirtualDevice interface abstracts the operations that apply to a multimedia stream device; the VirtualSwitch controls the manner in which VPI/VCI pairs are allocated and deallocated for ATM switches and ATM adapter cards. Support for QoS is realized via a set of abstractions that characterize the multiplexing capacity of networking and multimedia resources. The control of the ATM switches is accomplished via the *General Switch Management Protocol* (GSMP) [NEW96].

Figure 45 gives an overview of the current testbed architecture. At the lowest layer, the ATM LAN connects a set of hosts including multimedia devices (cameras, speakers, microphones and displays). The BIB is at the lowest software layer. *mcast* controllers, located in the layer above, make calls to the BIB interfaces to access the networking resources and multimedia devices. The highest layer contains the management entities.

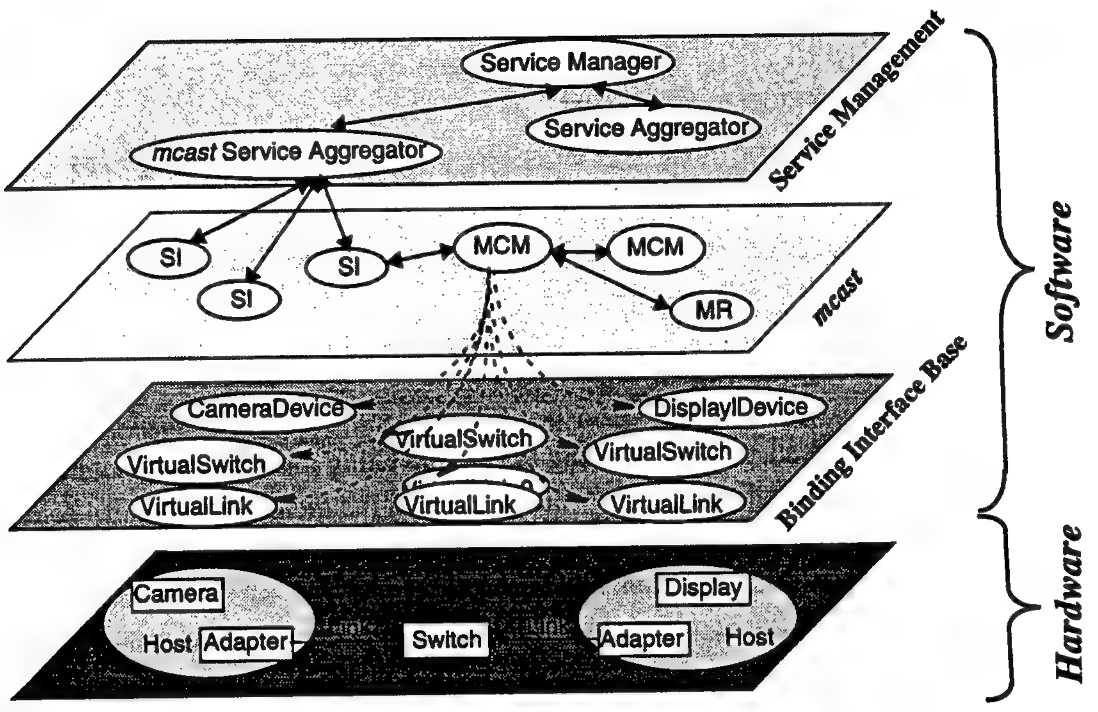
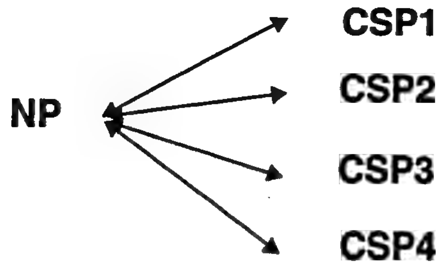


Figure 45 Hardware and software layers in the broadband testbed.

7.5 Peer to Peer Management

We envision a telecom environment in which several Connectivity Service Providers compete for network resources. One of them would be for example our *mcast* service provider. If that is the case the relationship among them would be as shown in Figure 46.



NP	Network Provider
CSP	Connectivity Service Provider

Figure 46

In the following we assume one CSP, namely our *mcast* SP and we focus on the interface between the NP and the CSP.

Peer SM to peer NM management

Management deals with configuring the system being managed and monitoring its utilization. Management is a feedback loop in which, based on the monitoring, control actions can be taken upon the system. Each management system in Figure 47 (NM and SM) deals with two main subsystems being managed: control and transport. Here we will only focus on the control subsystem. Figure 47 shows some of the objects we have defined in the control subsystems as part of our work.

Based on our definition of management above, we believe the information flow at the M interface should deal with both configuration and utilization issues.

- On one hand the Network Manager (e.g., the *xbind* NM) should make available to the Connectivity Service Manager (e.g., the *mcast* SM) the following information: network topology, capacity and operating point, and controller interfaces. *mcast* SM will use this information for different purposes, for example to configure the mcast controllers with the correct network controller interfaces so to realize the C interface.
- On the other hand, SM should be able to request (renegotiate) from NM specific resources for its own service (*mcast*).

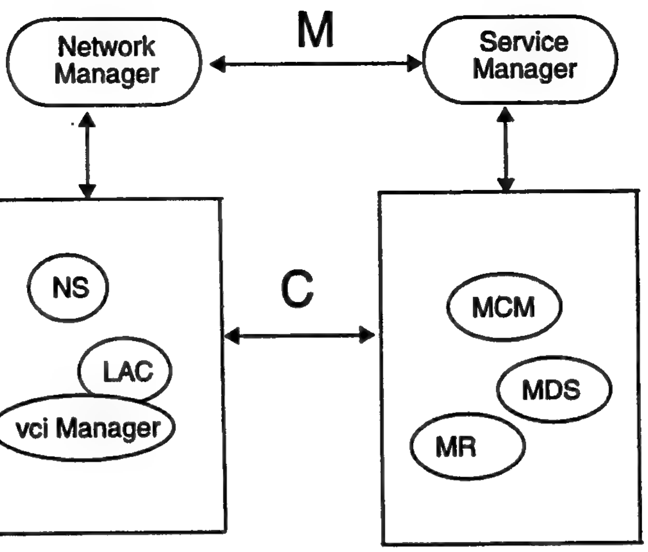
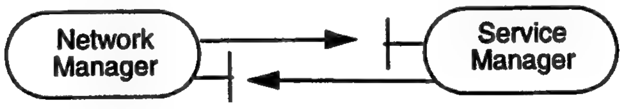


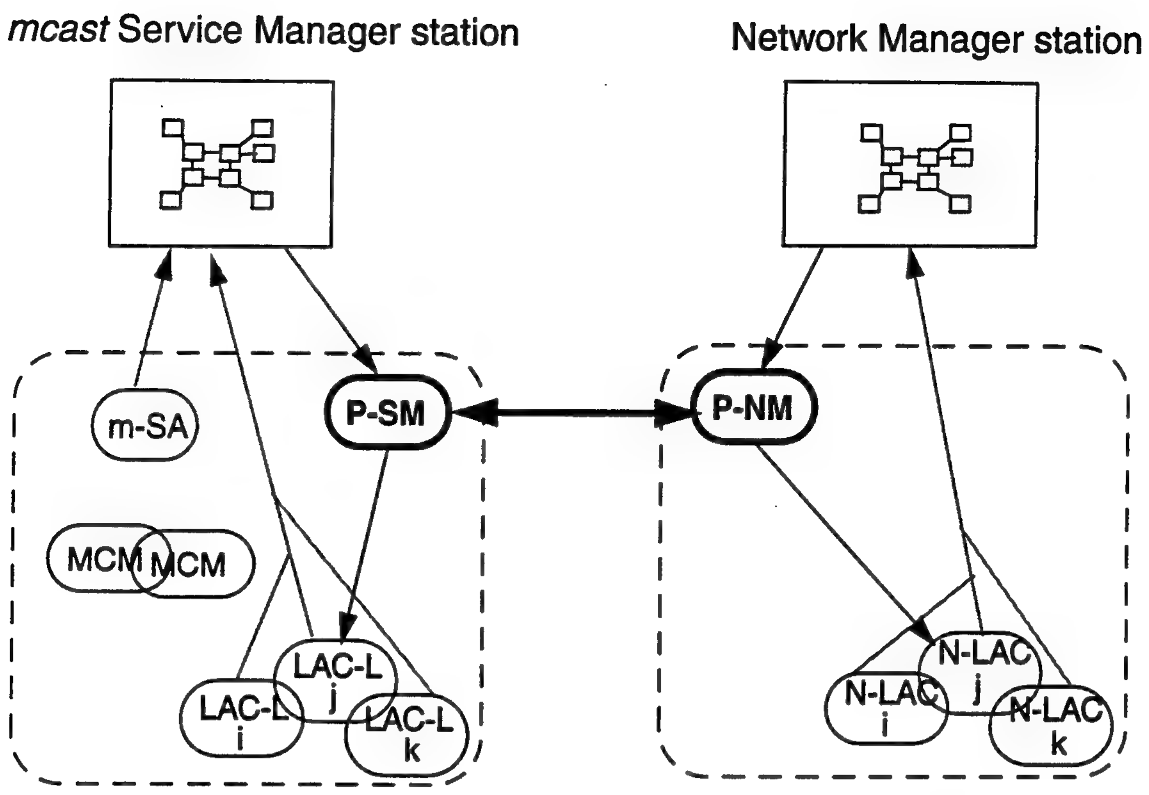
Figure 47

Scenarios

We envision to demonstrate the above concepts by using the following scenarios:

- A change in the data provided by NM triggers a change in SM's service. For example: A switch goes down or it is added to the network. This change will for example affect the view in the SM GUI.
- SM renegotiates with NM resources allocated to SM's service. More specifically, SM asks for more resources. SM asks for more capacity on a specific link. Upon completion of this operation -if the NP agrees to do so- the link capacity will be upgraded in the SM GUI.
- The interface M will be realized as two interfaces offered by the NM and the SM to each other, as shown in the Figure below.





P-SM is the Peer Service Manager, P-NM is the Peer Network Manager, m-SA is the *mcast* Service Aggregator, LAC represents the Link Admission Controller

Peer NM API

> NewBwReq (**in** linkId, **in** newBw, **in** newQos)
> NewBwRep (**in** linkId, **in** allocatedBw, **in** grantedQos)

Peer SM API

> NewBwNotif (**in** linkId, **in** newBw, **in** newQos)
> NewBwNotifAck(**in** linkId)

8 TeleSoft - Engineering and Evaluating Control and Management APIs

This chapter describes some of the concepts behind the TeleSoft prototyping platform and its implementation design.

TeleSoft applications are platform independent in two regards: independence of the system for which an application eventually is implemented (i.e. CORBA/UNIX environment.) But also independence whether the application runs on the real platform or as simulation only. In [CHA96a], a methodology for prototyping network protocols involving parallel simulation is introduced. TeleSoft combines the simulation approach with the real target platform: there is no need of implementing an application once for simulation purposes and once for every target system. By using the TeleSoft approach, the application will support all these platforms at the same time.

The software platform consists of two main building blocks: the API for the application programmer and the *adaptation layer*. The first part is platform independent; it allows to implement applications supported by multiple platforms. The adaptation layer on the contrary implements the functionality of the TeleSoft API on the target system. It is therefore dependent on the underlying system. A more detailed description is given in Section 8.3 , "Building Blocks".

The system currently supports CORBA ([OMG96]) for distributed applications on a 'real' target platform (TCP/IP, UNIX) as well as emulations based on the *Simulation Kernel* described in [PUT97]. These two systems will be addressed in Section 8.6 , "Adaptation onto the Simulation Kernel", and Section 8.7 , "Adaptation onto Orbix".

8.1 Introduction to TeleSoft

TeleSoft was designed to ease the task of prototyping telecommunication protocols. The platform attempts to provide a highly environment-independent API for implementing distributed systems. The goals are:

- to allow for fast prototyping of telecommunication protocols,
- achieve a high grade of environment independency,
- run tests on simulators without the need to deal with the complexity of real distributed networks,
- run scaling experiments involving topologies which are two expensive to be implemented on a real testbed on parallel supercomputers, and
- move the code, once the testing is done, without re-implementing or other major modifications from the simulator to the actual target platform and, if necessary, back.

The design of TeleSoft is driven by the above concepts. To understand Telesoft is important to study these concepts in detail.

8.1.1 TeleSoft — an Idea

TeleSoft is not only about simulation. Its power lies within the fact that TeleSoft applications are not only platform independent but also independent whether they run in a pure simulation environment or on an actual, real platform.

There are various possible target platforms for telecommunication protocols. The TeleSoft concept itself makes only two assumptions about such a platform: first, it must be programmable. Furthermore it must be powerful enough to implement the TeleSoft API. These are the only prerequisites a target platform must meet, whether it is a simulation or a 'real' target platform.

TeleSoft is not only a software system. It is a generally applicable concept, *a software design methodology.* It is, as will be shown, suitable for prototyping telecommunication networks. But it is also applicable in a wide range of other areas.

8.1.2 Motivation

The original motivation to develop this software was the following: after protocol prototypes for broadband multimedia networks have been developed using the simulator approach described in [CHA96a], the software had to be re-implemented for the testbed due to the significant differences between the two platforms. This slowed the process of prototyping down. TeleSoft was built to overcome this problem (Figure 48.)

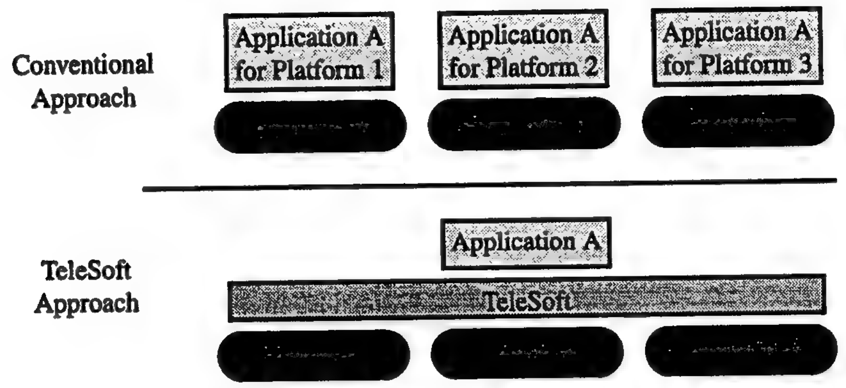

Conventional Approach
Application A for Platform 1
Application A for Platform 2
Application A for Platform 3
TeleSoft Approach
Application A
TeleSoft

Figure 48 The TeleSoft Approach avoids multiple implementations of the same application for different platforms.

TeleSoft is based on two technologies:

- Programmability: TeleSoft applications are pure software applications on a high level of abstraction. The underlying system must provide a sufficient level of programmability.
- Abstraction Capabilities of Object Oriented Programming: TeleSoft also needs the abstraction capabilities of OO languages. They allow a more simple and powerful design than conventional programming.

The independence of the platform is achieved by creating an abstraction of the involved platforms which is suitable for the task, in our case protocols for telecommunication protocols. The abstraction may be described as the concept of distributed software controllers and will be further described in Section "The Controller Model" on page 142. The step of finding an appropriate model is the most challenging one: on one side, the model's abstraction must be on a level high enough to include all platforms. On the other side it must also be specific and powerful enough to allow the application to be implemented. This abstraction is used to define an API which forms the framework for future applications.

A final step remains: in order to support a specific platform, the API needs to be implemented on it. What results is a software library for a specific platform providing a platform independent programming interface (see Figure 49.)

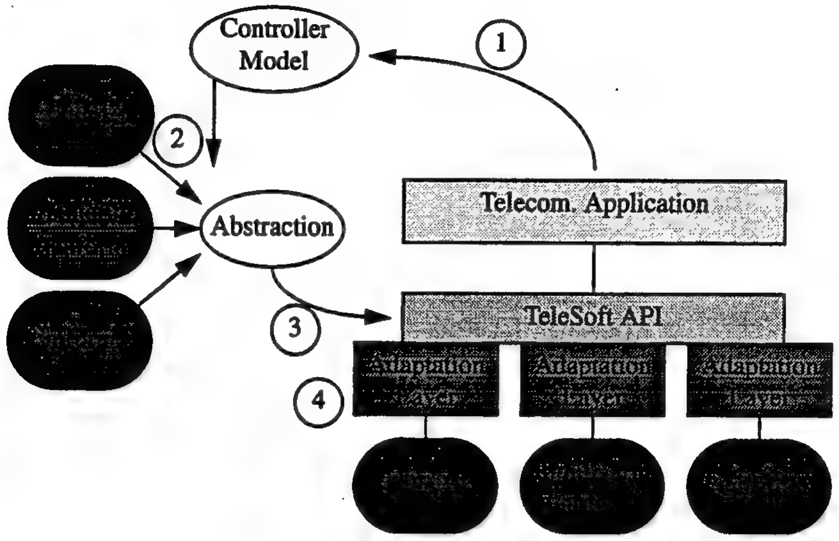


Figure 49 Development of the TeleSoft platform: first, a suitable model for the application had to be found (Controller Model, 1). Second, considering the requirements of all target platforms, the model was used to find a reasonable abstraction of the application (2.) From this abstraction, an API was derived. This API forms the framework for future applications. The final step was to implement the API for each platform (adaptation layer, 4.)

8.1.3 Using Simulation for Development

Simulation is widely used as a research tool. During the past years, a simulation platform, the Simulation Kernel, has been developed at the CTR. This system has been successfully used for different projects, such as a simulation of virtual private networks ([CHA97a] and [CHA97b]) and a multicast service protocol for broadband ATM networks ([AUR97a]). Its current implementation is described in [PUT97] and [PUT97a].

The Simulation Kernel implements a *Communication Network Simulator* based on the paradigm of *Discrete Event Simulation* (DES.) Each entity of the simulated network (later on referred to as controller) is implemented as independent object with its own local state (Figure 50). No global state information is allowed. Each of these entities communicates with others by sending asynchronous requests. Requests are queued at their destination and the controller is invoked (one of its application specific methods is called with the request as parameter) for each to allow its processing. While processing, the receiver has the possibility to generate further requests and send them to other controllers. The simulation is event (or request) driven: this means that if at any time no requests are left for processing, the system will halt. This is due to the fact that controllers may only generated new requests while processing one which was previously received. Figure 50 illustrates the concept of a Communication Network Simulator.

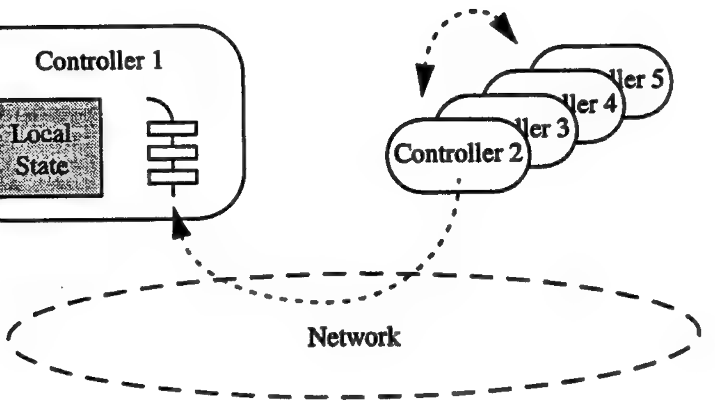


Figure 50 Communication Network Simulator: concept of distributed controllers, each with its own local state. Communication is only possible by sending asynchronous requests (events.)

Evolution of Time and Resources. One property of simulation in general is that the behavior of time usually doesn't correspond with the reality. Simulation is often used as a tool to eliminate time consuming or otherwise expensive real-world measurements. In that case, this behavior is certainly an advantage. But if the evolution of time influences the course of the simulation itself or even more is one of the variables being measured, problems arise. It is thus necessary to associate each simulated action with a time-stamp. This time stamp connects the simulation with reality: the goal is to determine those time stamps as accurate as possible.

This can be achieved in the following way: each action is associated with a time indicating how long it takes for the action to be completed. There are two possible actions: (A) the sending of a requests and (B) processing of requests. For the first, the time will indicate the communication delay which would occur, if the data was sent from the source to its destination in the real environment. For the latter, it indicates the time resources (cpu time) necessary for the computer to execute the commands associated with the request. Having this information, the simulation system is now able to determine the progress of time in a realistic manner by increasing the simulation clock for each action taking place in accordance to the user defined communication or processing delays.

Another issue is the following: if, for instance, the course of the simulation is visualized to allow for interactive monitoring and control through an interface. It would be highly disturbing for the human actor if the time advanced in a non-linear manner. Therefore it is necessary not only to allow batch simulations, but also experiments with a constant, user-defined ratio between actual and simulation time. Due to the discrete nature of the time model described in the previous paragraph, this is simple to accomplish; the processing of requests and hence the advance of time is restrained by adding idle intervals, in a fashion which forces the time to progress linearly according to a certain, predefined ratio.

Setting this ratio to one will make the simulation advancing in the same way as the application would in its real target environment. Thus the term 'real-time simulation.'

8.1.4 Introducing Parallel Simulation

In most cases, DES simulation is implemented serially, involving only one computer/cpu. In order to be able to simulate large network topologies and complex scenarios, the resources of single workstations proved to be insufficient.

To overcome this drawback, *PDES* (Parallel Discrete Event Simulation) was introduced. PDES allows essentially to run discrete event simulations on a distributed system. While gaining the power of a parallel computer, complete transparency over the complexity of such a system is maintained for the user. For further details about PDES refer to [FUJ90].

8.1.5 The Simulation Kernel as a Base for TeleSoft

The Simulation Kernel is a platform which allows the implementation of protocols for telecommunication or in general for any distributed system which can be separated into distinct entities. It supports miscellaneous UNIX platforms for serial simulation as well as parallel computers such as the IBM SP2 or the Intel Paragon for parallel simulation.

The design of the Simulation Kernel heavily influenced the design of TeleSoft. I.e. the monitoring and controlling mechanisms introduced with the Simulation Kernel are almost completely adopted for TeleSoft. Following a list of properties and features which can be found in both, the Simulation Kernel and TeleSoft.

- Global Time: the simulation conveniently provides a well-defined global time and so does TeleSoft. The accuracy on TeleSoft is limited though by the capabilities of the underlying system.
- Separate Entities (Controllers) with Local State: TeleSoft is implemented as communication network simulator. Though, as the following sections will outline, it is not limited to simulation only.
- Time Information: as described in above Section "Introducing Parallel Simulation", the DES simulation protocol demands a perfect a-priori knowledge of resource consumption (processing time for requests and communication delays for messages sent from one controller to another.) As will be described later in this chapter, TeleSoft applications running on the simulation platform has to provide this information to the system.
- Monitoring: often the user does not only want to run experiments for a certain period of time, but rather wants to monitor or even visualize state information. Therefore a generic mechanism for accessing a controllers state from an external tool has been introduced. This 'monitoring' mechanism has been carried over from the Simulation Kernel to TeleSoft, though it's implemented in a slightly different way.
- Controlling: similarly to monitoring, it's sometimes necessary to communicate parameter settings from an external source to a controller. This mechanism, originally introduced in the Simulation Kernel, has been ported to TeleSoft as well.

More background about earlier work and motivations leading to the Simulation Kernel can be found in [CHA95] and [CHA96a]. [PUT97] includes a detailed description of its implementation.

8.1.6 Moving on to the 'real stuff' — CORBA

In order to run an application not only on the simulator but also in a 'real' environment, it is necessary to introduce a system or environment suitable for telecommunication protocols.

An architecture which gained a lot of popularity during the last years is CORBA ([OMG96]). Some of the reasons for choosing CORBA are its flexibility and simplicity. CORBA also provides a relatively high level of abstraction and platform independence. It supports a C++ API, the language which was used for previous projects. And last but not least, the COMET group has been able to gather extensive experience with Orbix, the CORBA II implementation of IONA Technology.

CORBA is extremely powerful. It includes many features such as code marshalling and other. Most of these features were not used at all. The main purpose of CORBA as a TeleSoft target environment is to provide a simple platform for the communication within a distributed system.

8.1.7 Costs of Abstraction

Most platforms have certain advantages and certain disadvantages, certain features are supported while others are not. TeleSoft naturally cannot provide more features than a platform has. But, due to the abstraction, it also can only provide features which are supported by all target platforms involved. As a result. i.e. code marshalling is not available in the current implementation as it is only provided by CORBA, but not by the Simulation Kernel.

If the main field of application is prototyping, these drawbacks should be acceptable. The experience showed that the advantages overweigh by far. In addition, there is still the option of implementing code marshalling on the Simulation Kernel thus providing this benefit to TeleSoft as well.

8.2 The Controller Model

Next to the simulation approach, the most important part of the current TeleSoft implementation is the concept of the telecommunication controller and the controller API. Its design is (A) determined by the concept of a distributed system for telecommunication protocols previously developed in the COMET group, and (B) influenced by the design of the underlying platforms (Orbix and Simulation Kernel.)

The controller model assumes the following environment: a set of separated entities (controllers) dispersed among the nodes of a distributed system. In order to effectively implement a network protocol with this framework (i.e. multicast service) on any given platform (i.e. CORBA) certain prerequisites must be met.

- Communication capabilities: the environment must provide a facility for data exchange between separate entities.
- Knowledge of time: though not a necessity, it proved to simplify many tasks if a ‘global time’ is provided among all involved entities.:

- Ability to synchronize actions with time: often it is necessary to trigger events after a certain time interval or at a certain time. Timers provide a simple but yet powerful approach to deal with these problems.
- Naming: in order to distinguish different entities (i.e. for addressing), each controller must have a unique name.
- Locality: a telecommunication system is, needless to say, a distributed system. It is vital for each component to have the notion of locality, i.e. to know where in the network it is located and having access to the topology.
- External Monitoring and Control: certain tasks involve the interaction with external entities. Hence a controller should have the capability to receive data from and send data to an external, application specific source.
- Dynamic Controllers: networks not only involve dynamics in between of their components but also dynamics of the components themselves. On a slow time scale, new entities may be added or existing ones removed.

These prerequisites have so far proved to be sufficient for the purpose of implementing telecommunication protocols. The programming interface of TeleSoft, described in Section "The API" on page 144, was developed according to above model. Each of the concepts listed is provided to the application through a specific interface.

8.3 Building Blocks

The TeleSoft development environment consists of a software programming interface for the application (API) and a set of libraries. In contrary to the interface, there is one or multiple libraries for each platform. In order to run a TeleSoft application, the user has to provide one additional part, namely the actual controller implementation. By simply linking the controller implementation with a specific library, an executable for the chosen platform is produced. The controller code is independent of the library selected and hence platform independent (Figure 51.) Only the start-up procedure is platform and application dependent. It needs to be provided per platform and per application.

The building blocks of the TeleSoft software are:

- The TeleSoft API consisting of the abstract TS_Controller class.

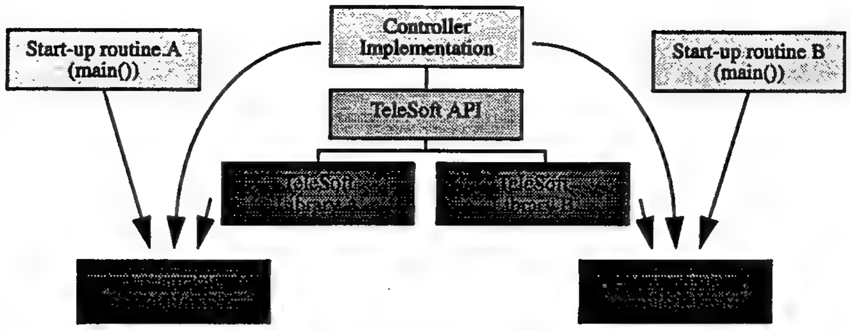

Start-up routine A (main())
Controller Implementation
Start-up routine B (main())
TeleSoft API
TeleSoft

Figure 51 TeleSoft Development Environment: the platform independent controller implementation is combined together with a platform dependent library to an executable for the chosen target platform. The start-up procedure though is also platform dependent and partly application dependent.

- An intermediate platform independent layer which includes some of the basic functionality of the TeleSoft platform and utilities.
- A final platform dependent adaptation layer implementing the API for a chosen system.

The first item, the API, is of most importance for the user. The API defines the framework an application has to be built on. The third item implements the functionality which is provided by the API on a specific platform. The intermediate layer defines an object oriented scheme for the mapping of the API to the adaptation layer. Though it consists of few lines of code only, it is here were the basic approach to achieve platform independence is implemented. Figure 52 on page 145 shows the different layers and how they are connected to the application and to the target platform respectively.

Each of the three TeleSoft software layers is covered by one of the following sections starting with a description of the API.

8.4 The API

The API bases on the controller model for telecommunication protocols introduced in Section "The Controller Model" on page 142. The API itself is platform independent. But in order to support a specific platform it must be implemented thereon. The implementation of the API on this

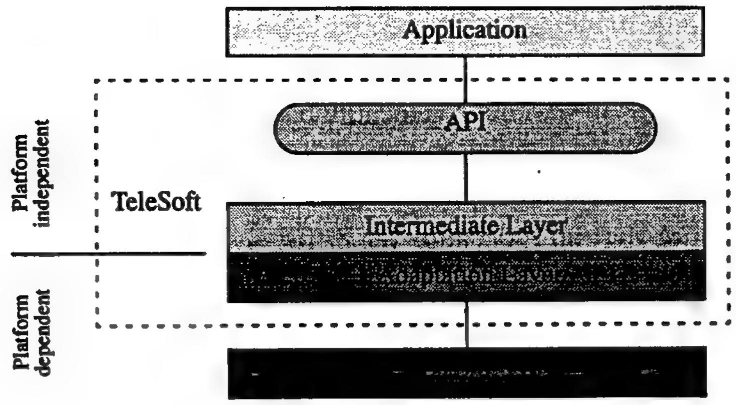


Figure 52 Building Blocks

platform must of course be feasible. Hence it is necessary to take the design, i.e. the set of supported features, of each of the target platforms into account.

The influence of each of the target platform's design on the API is only of technical nature. It does not affect the controller model itself. Nevertheless it is helpful to know about these issues in order to understand the design of the API.

TeleSoft currently supports two target platforms; the Simulation Kernel serving as an emulation platform and CORBA as one possible choice of a 'real' target platform. Naturally, the amount of features provided by CORBA exceeds the features offered by the rather simple Simulation Kernel. Hence, in favor of the support of the Simulation Kernel, many features provided by CORBA are not deployed by TeleSoft.

The following is a list of issues which need to be considered for the design to support both platforms.

- OO Design: due to the (C++) object oriented nature of both, the Simulation Kernel and Orbix, it is the natural solution to design the interface for the controller by defining a C++ base class.
- No Code Marshalling, No Synchronous Communication: CORBA has a number of powerful features; extension of a object's interface by using inheritance, typed arguments and code marshalling, synchronous (RPC like) calls to remote controllers etc. TeleSoft cannot use these features as they are not supported by the Simulation Kernel. On the Kernel, one

predefined method is invoked whenever requests are sent to a controller. The method's declaration cannot be modified by the user. Futhermore are the parameters not typed but must rather be encoded into a byte stream.

- Concept of server: CORBA defines locality by means of servers (and hosts.) Though not directly provided by the Simulation Kernel, it was easy to map CORBA's hierarchical onto TeleSoft's non-hierarchical naming scheme.
- Platform specific Start-up Procedure: the difference between starting executables on a parallel supercomputer for simulation as opposed to starting CORBA servers on a distributed network makes it virtually impossible to define a common start-up procedure. Therefore the starting and initialization step is intentionally left undefined and must be provided by the user for each platform and application.

8.4.1 The Class Interface

The model introduced in Section "The Controller Model" on page 142 and the design of the two platforms which needed to be supported resulted in the controller interface as shown in Figure 53. It is defined by the public and protected part of an abstract C++ class: one part of the interface declares virtual methods which must be defined by the application. I.e. the method processRequest() is invoked for each arriving request. The second part of the interface provides functionality necessary to the application. I.e. getTime() returns the current time.

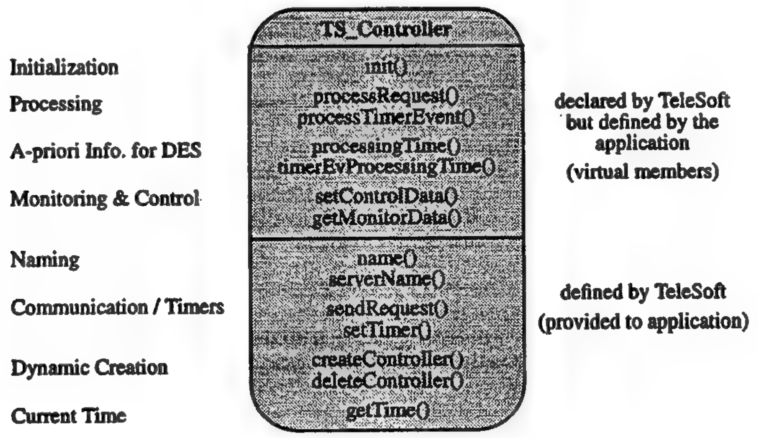


Figure 53 Controller Design/API: the controller is implemented as C++ class, concrete application controllers inherit the interface by deriving from it.

In order to implement an application controller, the user needs to defined a C++ class publicly deriving from the TS_Controller class. By redefining (overloading) the virtual methods, the application specific semantics are implemented.

Table 1 lists all methods, their in- and output parameters which need to be defined by the derived (application) class.

Method Name	Input	Output	Semantics
init()	user data		Initialization
processRequest()	user data		Invoked for each arriving request
processTimerEvent()	user data		Invoked for each previously set timer as soon as it expires
processingTime()		time	Returns the time the controller needs to process a request
timerEvProcessingTime()		time	Returns the time the controller needs to process a timer event
setControlData()	user data		Invoked when control data from an external source is received
getMonitorData()		user data	Invoked when monitoring data is requested from the controller

Table 1: TS_Controller virtual methods: this methods need to be overloaded for each application defined controller class.

Input and output parameters of type 'user data' are implemented as byte stream. The semantics (length and content) is application dependent.

Table 2 lists the methods which provide necessary functionality to the application controller (i.e. for sending request etc.) Note that these methods are platform specific and hence implemented for each platform in the respecting adaptation layer (TS_SysObject.)

Method Name	Input	Output	Semantics
name()		string	Returns the name of the controller
serverName()		string	Returns the name of the local server
sendRequest()	user data		Sending requests to a remote controller
setTimer()	use data, time		Sets a timer event

Table 2: TS_Controller predefined methods providing necessary functionality to the application.

Method Name	Input	Output	Semantics
createController()	user data		Dynamical create new controllers
deleteController()	time		Dynamically removes an existing controller
getTime()		time	Returns the current (global) time

Table 2: TS_Controller predefined methods providing necessary functionality to the application.

Some of the methods have slightly different semantics depending on the platform. For example the method getTime(); it returns the 'global' time. The Simulation Kernel has a built-in mechanism to synchronize the simulation clocks between different processes. In contrast to this, there is no such mechanism provided by CORBA. The implementation of the adaptation layer for CORBA makes the assumption that the system clock (gettimeofday()) provides this time accurately. The system clock on UNIX platforms is usually synchronized within local networks with either timed (time daemon), timeslave (time slave) or nptd (network time protocol daemon.) But measurements indicated that the divergence between two hosts on one LAN using i.e. ntpd for synchronization can exceed 5 or 10 milliseconds dependent on the system load, network load and other factors. Hence, while on the Simulation Kernel this 'global' time is guaranteed to be correct, this is not the case on CORBA.

The current implementation of TeleSoft, being a prototype itself, does not clearly define the necessary behavior for each method. In future work extending the software this is definitively a conceptual and implementation aspect which should be improved.

For more details and explanation on how to implement TeleSoft applications, refer to the TeleSoft user manual.

8.4.2 The Intermediate Layer

The intermediate layer contains utility classes for the API as i.e. a string class, the TS_Message class for storing user-data and other. But its main purpose is to define a general scheme for implementing the API on a specific platform. It is the structure of this layer which makes the platform independent programming interface possible at all.

8.4.3 The Platform Interface

While the API discussed in the last section provides an interface upwards or towards the application, another class named TS_SysObject provides an interface downwards, or towards the platform. Both of these interfaces serve the same purpose; hiding implementation specific details. In contrast to the controller API which unifies all possible applications (or application controllers) with one common interface, the TS_SysObject interface abstracts the platform.

To add support for a new platform to TeleSoft, the interface of the TS_SysObject abstract base class is to be implemented in a derived class. This scheme corresponds to how application controllers are derived from the base class TS_Controller. The TS_SysObject's interface has two parts (Figure 54.) One part includes the virtual methods which will provide the basic functionality

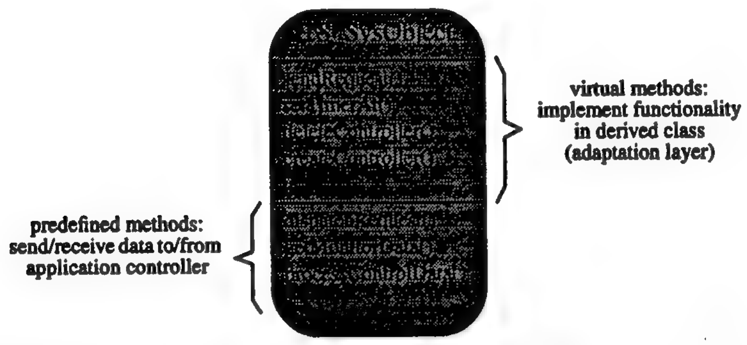

TS_SysObject
virtual methods: implement functionality in derived class (adaptation layer)
predefined methods: send/receive data to/from application controller

Figure 54 interface of class TS_SYSOBJECT serving as abstraction for the platform (incomplete.)

of TeleSoft. They need to be implemented specifically for each platform (in each class derived from TS_SysObject.) I.e. the sendRequest() method accepts user data as argument and sends it to the destination controller. There are no specifications on how these semantics are implemented. For actual examples refer to the following two sections, "Adaptation onto the Simulation Kernel" on page 154 and "Adaptation onto Orbix" on page 155.

The second part of the interface is already implemented in the base class to prevent code duplication in the subclasses. For instance, the method dispatchRequest() forwards an arriving request to the application controller for processing.

8.4.4 Mapping TS_Controller to TS_SysObject

To combine the TS_Controller interface with a specific TS_SysObject implementation, a design pattern referred to as bridge pattern by [GAM95] is used (Figure 55.) The bridge pattern is used to

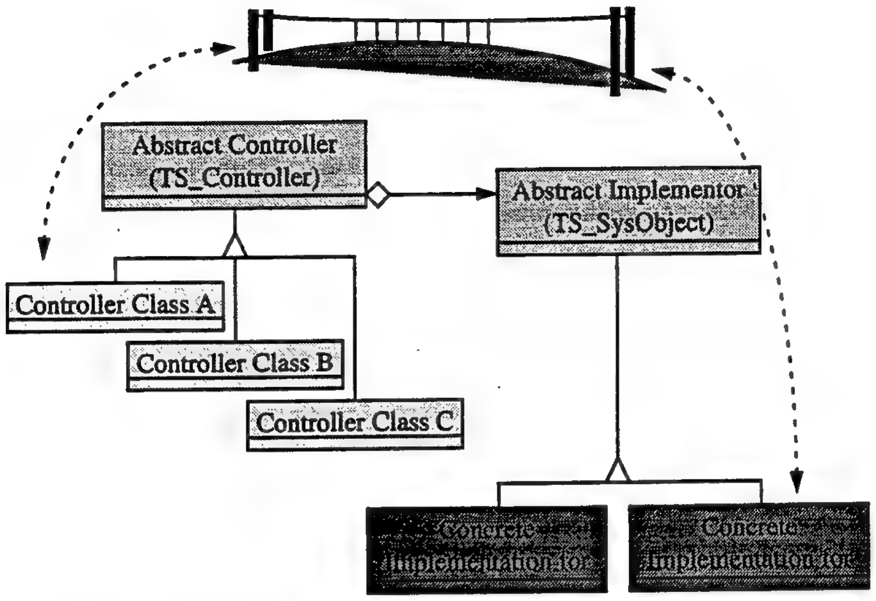


Figure 55 Bridge Pattern: the basic functionality available to the controller is implemented in a class derived from the abstract implementor (bridge) class TS_SysObject. This needs to be done only once for each platform.

de-couple an abstraction from its implementation so that the two can vary independently. In TeleSoft, the class TS_Controller is the abstraction (for application controllers), while TS_SysObject is the implementation of that abstraction (on a specific platform.) A class serving as bridge is usually represented by an abstract base class (abstract implementor, TS_SysObject.) Its interface provides a certain functionality to the abstraction (abstract controller, TS_Controller.) The actual implementation of both is provided in subclasses. The entities of each of the (derived) classes are completely interchangeable even at run-time. The implementation details (concrete implementor) are hidden within the concrete implementor.

At run-time, each application controller (derived from TS_Controller) is combined with a instance of the implementor (class derived from TS_SysObject.) Depending on which platform the application is running on, a different implementor is chosen. The type of application controller of course depends on the user's specifications. The actual implementation (TS_SysObjectImpl) is now able to reach the application controller or rather its interface through the methods defined in the base class TS_Controller, independent of how they are actually implemented. On the other hand, the TS_Controller can use the functionality implemented for a specific platform through the TS_SysObject public interface, again independently from its implementation details.

The internal design of the intermediate layer is not visible by the application. This is achieved through copying certain methods from the TS_SysObject's to the TS_Controller's interface. I.e. for sending requests, the application code does not need to access the underlying TS_SysObject's interface. Rather it invokes a methods predefined in the TS_Controller interface (TS_Controller::sendRequest().) This function simply forwards the request by invoking the respecting method of the TS_SysObject interface (TS_SysObject::sendRequest()) on the respecting object. Figure 56 shows an example invocation sequence for incoming and outgoing requests where the application is running on CORBA. The left box represents the application controller, the right box the TS_SysObject implementation. The innermost bar depicts the interface of the abstract base class, the outermost bar the interface of the actual implementation within a derived class. The figure illustrates how the interfaces of the base classes are exposed to each other while the actual implementations within the derived classes are not.

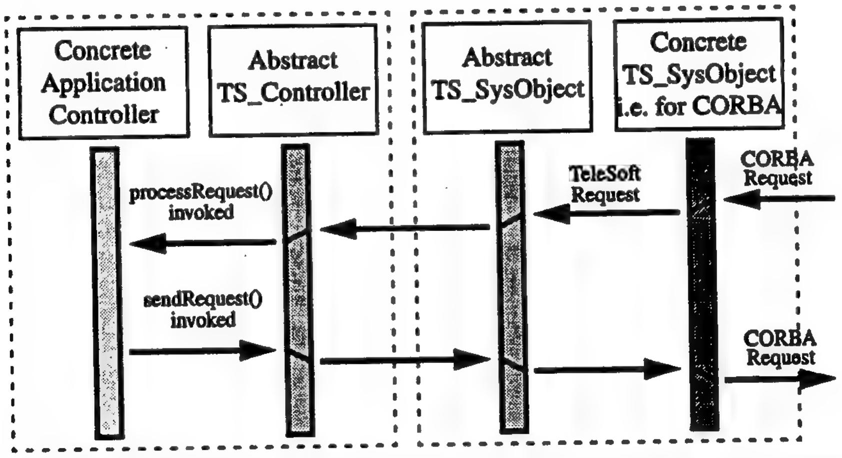


Figure 56 TeleSoft Invocation Sequence: incoming requests are processed by the TS_SysObject Implementation according to platform specific requirements. The data is then passed to the TS_Controller class via methods implemented in the TS_SysObject base-class and is finally specified as an argument to the processRequest() method of the concrete controller. When sending requests, the invocation sequence is similar except that the data flows in opposite direction.

The bridge scheme has two major advantages:

1. The application controllers (classes derived from the TS_Controller class) can be refined according to the user's needs independently of the specific TS_SysObject implementation (subclass), hence independent of any platform specific details.
2. Flexibility for embedding the concrete TS_SysObject implementation into the target platform.

This approach simplifies the task of implementing a software controller for multiple platforms enormously. Instead of implementing N controller for each of M target platforms resulting in NM objects, this approach allows the same task to be achieved with $N+M$ objects only.

Figure 57 shows once more an overview of the software design of TeleSoft and its layers including the classes implemented there in.

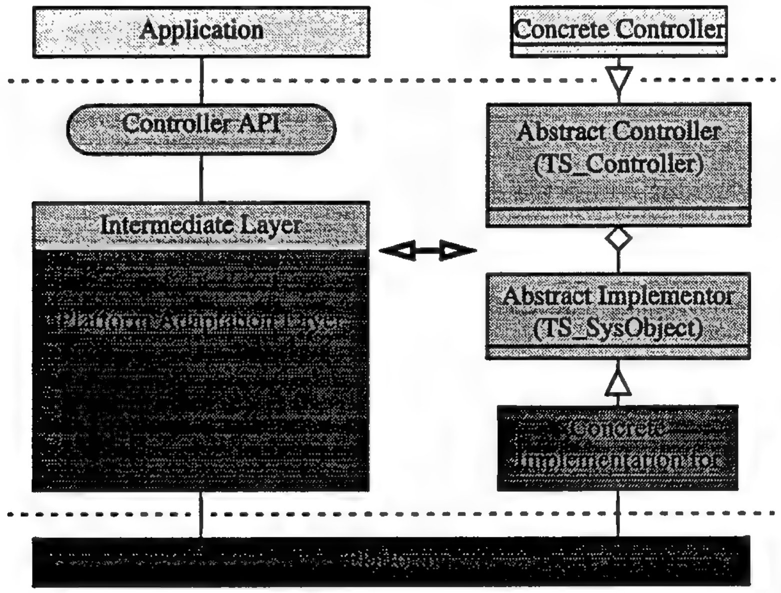

Application
Concrete Controller
Controller API
Abstract Controller (TS_Controller)
Intermediate Layer
Platform Adaptation Layer
Abstract Implementor (TS_SysObject)
Concrete Implementation for

Figure 57 TeleSoft Software Design: while the class TS_Controller provides the API for the user, the mapping onto the platform is established by the implementation of the 'bridge' class TS_SysObject.

8.5 The Adaptation Layer

The adaptation layer consists of a platform specific specialization of the TS_SysObject class. The following four functional components must be implemented for each platform: sending requests, time, timers and dynamic controller creation and removal. Table 3 lists the respecting methods which need to be implemented.

Method	Input	Output	Functionality
sendRequest()	user data		send a request and invoke the processRequest() method on the given controller
setTimerAt()	time		trigger a timer event at the calling controller at the given time (invoke processTimerEvent() at that time)
createObject()	ctrl. type		create a controller of given type and add it to the set of application controllers
deleteObject()	ctrl. name		delete the specified controller from the current set of application controllers
getTime()		time	return the current time

Table 3: TS_SysObject virtual methods: this functionality must be implemented for each platform using the above interface.

In addition to those functional components, it is also necessary to map the data types and data formats used by the TeleSoft interface to the corresponding types and formats provided by the platform. I.e., within the TeleSoft API, the data type used for time is a non-negative, 32-bit integer. If the underlying platform does not provide a similar representation of time, a mapping has to be defined. Table 4 lists the TeleSoft data types and their application:

Data Type	Format	Usage
TS_CtrlName	string	name for controllers and servers
TS_Timer	long	as time specifier
TS_Message	byte stream	wraps user data

Table 4: Data Types used within the TeleSoft API

It should be noted that, in the current implementation, the calling sequences within the adaptation layer between the two classes TS_Controller and TS_SysObject are neither carefully determined nor clearly defined. There is certainly room for improvement.

The following two sections give an outline on how the adaptation layer was implemented for supporting the Simulation Kernel and CORBA.

8.6 Adaptation onto the Simulation Kernel

The adaptation of TeleSoft onto the Simulation Kernel proved to be fairly simple as the interface structure of the Simulation Kernel and TeleSoft are very similar.

The design of the Simulation Kernel's API is the following: each entity is derived from the class SimulationObject. Virtual methods which must be defined include ProcessEvent(), GetProcessingTime() etc., the similarities are quite obvious. For more details on the Simulation Kernel's programming interface refer to [PUT97a].

The mapping between the TS_SysObject class and the SimulationObject class was established by using multiple inheritance. This allows to combine their functionality and interfaces (see Figure 58.)

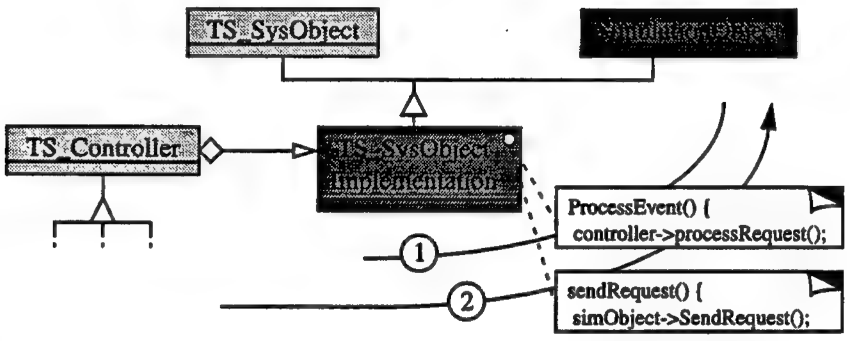

TS_SysObject
SimulationObject
TS_Controller
TS_SysObject_Implementation
ProcessEvent() {
controller->processRequest();
1
sendRequest() {
simObject->SendRequest();
2

Figure 58 Adaptation Layer for Simulation Kernel: the concrete implementation of the TS_SysObject combines its two base classes through multiple inheritance. (1) illustrates how requests passed from the platform to the SimulationObject are simply forwarded to the controller. When a controller sends requests, the data is passed in opposite directions to the Simulation Kernel for delivery.

Some additional notes about how the implementation of the functional components:

- Time: the 'time' provided by the Simulation Kernel and the time used within TeleSoft are similar; both define it as a positive integer. Hence there is no conversion necessary to obtain the 'current' time from the Simulation Kernel and provide it to TeleSoft.

- Mapping Controller to Processor Nodes: when the kernel is used for parallel simulation, typically more than one node is involved. The distribution of controllers among all nodes is transparent for the application. For efficiency reasons however it is necessary to allow the user to define a specific mapping (static load balancing.) For this purpose, a utility has been added to the adaptation layer.
- Note that a node does not correspond to a server: a server is a aggregation of controllers on a network node. A node in this context refers to one processor out of a pool used for the parallel simulation. It does not have a meaning within the network or controller model. Within a simulation, the assignments of controllers to network nodes can be ignored (though it is logically still valid.)
- Naming Conventions: the Simulation Kernel identifies every object by a character string with maximum length of 32. TeleSoft uses zero-terminated strings of any length. For simplicity reasons, it is assumed that the maximum length used by the application is 32. This makes the mapping of Simulation Kernel identifiers to TeleSoft identifiers trivial. Server names as defined in the TeleSoft API, are ignored.
- Communication: when a request is sent from one controller to the other, the data is simply encoded into a byte stream and forwarded by using the respecting functionality of the Simulation Kernel (SimulationKernel::SendRequest().)
- Timer: though timers are not directly supported by the Simulation Kernel, they can easily be 'emulated': the Simulation Kernel needs the application to specify a communication delay with each request sent. This is the time the request will be delayed before it's delivered to the receiver's queue. By specifying a communication delay which equals the difference between the desired time for the timer to elapse and the time it is set, this mechanism can be used for implementing timers. The requests sent to trigger timer events are marked with a flag to allow the distinction from regular. As destination, the controller setting the timer is specified. Once the time is elapsed, the request is returned to the sender indicating the timer even.
- However, it is important to note the difference between these timers and the ones implemented in the adaptation for Orbix. While, in the Orbix implementation, timers have a higher priority as requests, on the Simulation Kernel they haven't; they are queued in the same queue as all other, regular requests. Therefore, the longer the queue, the less accurately timer events are triggered.

8.7 Adaptation onto Orbix

The adaptation layer for Orbix proved to be more difficult to implement. This was mainly due to the CORBA's synchronous nature. While asynchronous calls are supported through oneway methods, synchronous interactions between servers still take place. Unfortunately, Orbix proved to be very susceptible to dead locks when used for asynchronous message passing.

TeleSoft uses IONA Technology's CORBA II compliant product Orbix. A few of their proprietary extensions allowed us to deal with issues such as dead-lock prevention and time-outs.

8.7.1 The CORBA Object and its IDL Interface

The base for the controller when running in a CORBA environment consists of the CORBA object. This object establishes the link between the TS_Controller/TS_SysObject couple and the CORBA programming interface: it provides three components: (1) the data type for exchanging user data, (2) a oneway method for controller initialization and (3) a oneway method providing the interface for sending requests. Its interface is as shown below.

```
const short MAX_MESSAGE_LENGTH = 2048;

typedef struct OrbixMessage_ {
  octet data[MAX_MESSAGE_LENGTH];
  short dataLength;
} OrbixMessage;

interface OrbixObject {
  oneway void processEvent(in OrbixMessage msg);
  oneway void start(in long sec, in long usec);
};
```

The IDL interface is translated using the BOA approach (see [ION97] for details) resulting in the C++ class OrbixObjectBOAImpl. It provides a communication interface for other controllers.

8.7.2 Coupling CORBA Object and TS_SysObject

The TS_Controller class and the CORBA object are linked together using a similar scheme as in the adaptation layer for the Simulation Kernel. The concrete implementation of the TS_SysObject class (TS_SysObjectImpl) derives not only from that base class, but also from the CORBA object OrbixObjectBOAImpl. It combines the capabilities of the CORBA object (receiving and sending CORBA requests) with the interface of the class TS_SysObject (Figure 59.)

At run-time, each application controller is linked to one instance of class TS_SysObjectImpl. The latter receives CORBA requests through the CORBA method

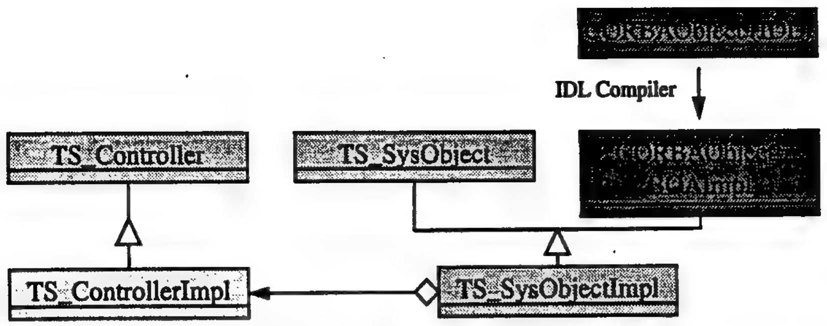


Figure 59 Coupling CORBA with the TS_Controller: each application controller is linked with on instance of class TS_SysObjectImpl. The latter combines the capabilities of a CORBA object with the interface of its second baseclass TS_SysObject.

TS_SysObjectImpl::processRequest() and forwards them to the controller for processing. New requests generated by the application controller are sent to the destination by invoking the processRequest() CORBA method of the TS_SysObjectImpl instance associated with the destination controller (illustrated in Figure 60.) See also Section "Mapping TS_Controller to TS_SysObject" on page 150 for a general description of this approach.

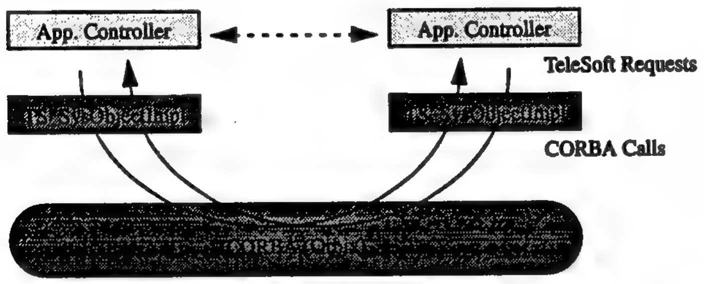


Figure 60 Communication using CORBA: the requests are sent from one controller to the other by using the OrbixObject (TS_SysObjectImpl) as interface.

In the current implementation however, the communication takes never place between the sending and the receiving controller directly. Each server includes one non-application object, the 'creator' object which is responsible for the 'administrative' tasks. It receives all requests from either local or remote controllers and forwards them to the destination for processing. It also creates new controllers if requested. Note that this 'creator' is not shown in Figure 60.

8.7.3 CORBA Servers

Each application controller is associated with one server and each server runs on a specific host (Figure 61.). Both, the mapping of controllers to servers and the distribution of servers among the

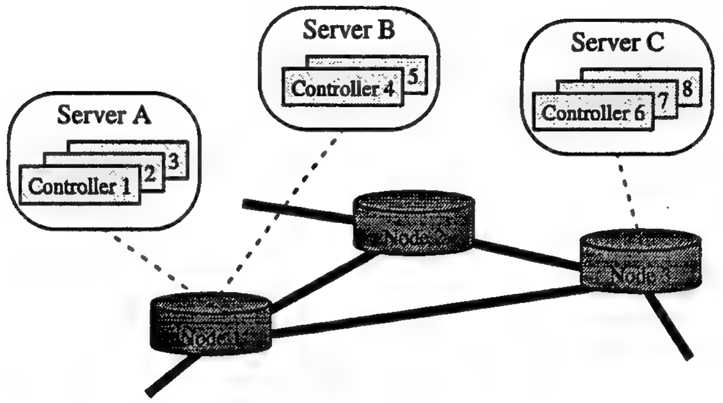


Figure 61 Controller to Server and Server to Host Mapping.

nodes is defined by the user. The first in part of the TeleSoft application itself (usually in a configuration file), the latter is configured through CORBA/Orbix.

8.7.4 Inter-server vs. Intra-server Communication

Each request, whether its destination is local to the sender (allocated on the same server) or on a remote host, is sent to the creator object off the destination server through a CORBA call (Orbix-ObjectBOAImpl::processEvent().) CORBA hides the location of an object. Hence it does not matter whether a request is actually sent to a controller allocated on a remote server or even host or not.

This transparency provided by CORBA first caused a serious problem. A basic assumptions made by TeleSoft is the following: each request is processed atomically (at least from the controller's point of view.) It must be guaranteed that the processing is not interrupted, at least not by actions which change the state of the controller. For instance, recursive invocation of the processRequest() method are not acceptable.

CORBA oneway calls however are processed synchronously if the destination object resides within the same server (quite naturally as CORBA calls model method invocations.) While for most CORBA applications this behavior is desired, in case of TeleSoft it is not. Consider the case where a controller sends a confirmed request to a neighboring (local) controller. The other controller, residing within the same server, processes the request immediately and sends the reply back. The reply in turn triggers the invocation of the processRequest() method of the controller which originally sent the request. That controller however did not yet terminate the processing of the original request - the original request is not executed atomically.

Controller code which must be able to deal with such recursive invocation would be much harder and more complex to implement. To avoid the need for it, each server has a limit of one request which is processed at a time. If new requests are generated internally or otherwise arrive during processing, they are queued and processed as soon as the server is idle.

8.7.5 Dead-locks during Communication

When a CORBA object is invoking a method on a remote object, the data is communicated synchronously and the sender is blocked until the receiver accepted the data (RPC semantics.) Even when using one-way calls, the data is communicated synchronously possibly resulting in the sender being blocked.

This scheme can lead to various deadlocks scenarios. For instance, if three servers try to contact each other circularly at the same time (server A invokes a request on B, server B on C and server C on A,) a dead-lock occurs. To avoid them, a time-out has been introduced limiting the time servers are blocked. If the time-out expires, the request is temporarily queued. Later on a new attempt is made to deliver it.

8.7.6 Time and Timers

Timers are implemented as follows: each controller has a timer queue where one item is stored for each timer set. Every list-item carries a time stamp indicating the time it expires. Each server checks regularly for expired timers and processes them if they occur. See also Section "The Event Loop" on page 159.

8.7.7 The Event Loop

Once a server is launched and has initialized all its controllers, it suspends execution and waits for incoming requests. Once arrived, they are forwarded to the controllers for processing. After that, the server returns to its idle state and awaits further requests. We refer to this loop of continuously waiting for requests and processing them as 'event loop'.

Besides of regular requests, controller must also process previously triggered timer events. Whenever a controller sets a timer event, one item containing the expiration time is appended to a queue. Once it is found that the current time passed the once indicated by one of the items, it is removed and the controller is timer event is triggered by invoking the controller's method processTimerEvent(). In order to be able to regularly check the timer queue for such expired events, the default event loop provided by Orbix has been extended.

```
int
OrbixStartup::enterLoop(char* serverName)
{
   ...
   while (1) {
      TS_SysObjectImpl::checkTimerQueues();
      CORBA::Orbix.processNextEvent(Process_Timeout);
      processMsgQueue();
   }
}
```

The method checkTimerQueues() scans the timer queue of each controller and triggers expired events. The method processNextEvent() is supplied by Orbix; it processes any incoming requests for the given period of time. The last statement attempts to re-deliver requests for which the previous sending timed out (see Section "Dead-locks during Communication" on page 159.)

8.7.8 Dynamic Object Creation and Removal

Dynamic objects are created by the 'creator' object introduced in Section "CORBA Servers" on page 158. When a controller issues the request to create a new controller of a given type, it is forwarded to the creator object of the server where the new controller is assigned to. The controller is constructed there by invoking a user-supplied function (create_controller()) which returns a pointer to the new object.

The request to delete a controller may only be issued by that controller itself. It is processed within the adaptation layer by simply deleting the controller and the associated TS_SysObject.

8.7.9 Server Registration

For inter-server communication, Orbix must be able to resolve the host where a destination server is located. The *server to host mapping* is part of the Orbix server configuration and registration.

Servers can be registered in either persistent or non-persistent mode. Persistent servers must be invoked manual while non-persistent ones are launched automatically be orbix during the first bind request. TeleSoft servers are persistent. Hence it's best to also register them in persistent mode.

For details on how to configure Orbix and the use of related utilities like putit and server-hosts, refer to the Orbix Programming Manual [ION97].

8.7.10 Dead-locks during Binding

Binding refers to the initial connection set-up to a remote CORBA object before invocation of its methods. The binding creates and initializes a pointer to that object. The binding phase unfortunately introduces another source for dead-locks: namely, if two servers concurrently try to bind to each other, they are blocked indefinitely.

Dead-locks while binding are much less frequent than dead-locks occurring due to mutual or circular method invocation described in Section "Dead-locks during Communication" on page 159. They are more likely to occur shortly after the application starts. It is however more difficult to avoid them: while Orbix provides functionality for specifying time-outs valid for binding, their implementation is not reliable. It can occur that a TeleSoft/CORBA application dead-locks and must be restarted.

8.7.11 Application Initialization and Start-up

Launching TeleSoft application on Orbix involves two steps: first, all server must be started. Each of them creates the set of controller which has been assigned to it.

Once all the servers have been launched, a CORBA client is executed. This client binds to each server's creator object and invokes its start() method. As argument, the current (system) time is specified. This start-up time is used to calculate the current time as returned by TS_Controller::getTime(). It is always relative to the start-up time allowing a global time beginning at zero. The accuracy of this 'global' time is of course limited by the accuracy of the clock synchronization between the hosts involved.

8.7.12 Application Exit

No graceful application termination is currently supported. It is currently accomplished by killing all participating processes (the CORBA servers.)

8.8 Start-up Procedures

The start-up of a TeleSoft application is the only part which cannot be realized in a system independent manner. The sequence in which controllers are created and initialized however is strictly defined. How this sequence is implemented for a specific TeleSoft platform such as CORBA or the Simulation Kernel does not matter.

1. The constructor of the TS_Controller (base) class is invoked with the arguments *controller name* and *server name*. This may or may not be followed by invocations of further class specific constructors.
2. The virtual method init() is invoked specifying any optional initialization data.
3. Only now incoming requests may be passed to the controller for processing.

Once the init() method is invoked, the controller is allowed to send requests and set timers. The controller should not attempt to send request or set timers while being constructed (during the constructor call.)

8.9 Interaction with External Applications

TeleSoft application live in a closed world. While each controller may interact with others, they are separated from the underlying system including any other applications running thereon. This kind of self-contained applications however are rarely of any use. In order to actually perform some kind of task, systems need to interact with their environment in some fashion.

The original Simulation Kernel introduced a simple but generic way for interaction: the *monitoring and controlling*. In spite of its simplicity, this facility proved to be a powerful approach for implementing an interface between the TeleSoft application on the one side, and visualization, monitoring and controlling applications on the other. Later on, while gaining more experience by working with applications, it became clear that additional ways for interaction with external applications were necessary. In the current TeleSoft implementation, three different levels of interaction can be distinguished. Two of them are part of the TeleSoft API.

8.9.1 Indirect, platform independent interaction

TeleSoft defines a external interface for interaction with one or multiple remote applications, the *monitoring and controlling* interface. It is implemented as a client/server scheme where the remote application acts as client and one specific controller as server. The server controller accepts commands through a TCP/IP connection, processes them and optionally sends back a reply. Two different kinds of requests exist, either sending control information to, or requesting status information from a controller. Controlling data is provided to a controller through invoking one of its methods (TS_Controller::setControlData().) On the other hand, the method getMonitoringData() is used to collect status information which eventually is sent back to the requestor (see illustration in Figure 62, part A.) The use of controllers participating in this kind of interaction is not restricted to a certain platform (i.e. CORBA or Simulation), hence the name 'platform independent interaction.'

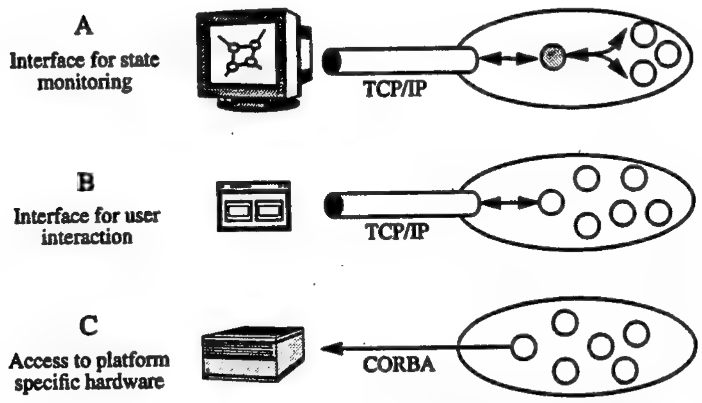

A
Interface for state monitoring
TCP/IP
B
Interface for user interaction
TCP/IP
C
Access to platform specific hardware
CORBA

Figure 62 Interactions with external applications: TeleSoft provides 2 generic facilities (A and B). But it is also possible to use platform specific features as in (C) where a CORBA call is made from within the TS_Controller::processRequest() method.

One of the applications which has been developed using the TeleSoft approach is a multicast protocol for broadband ATM networks. Using the above monitoring facility, the status (active calls etc.) for each link is communicated to an external interface and visualized. Additionally, certain protocol specific parameters may be set by sending control information from the visualizing application to TeleSoft controllers.

8.9.2 Direct, platform independent interaction

TeleSoft also defines a interface for direct communication with a specific controller. In addition to the usual abilities, these controllers are able to receive and send data from or to an external source. In contrary to monitoring and controlling, this facility is designed to provide an interface to one specific controller, not to the application (Figure 62, part B) as a whole. A controller gains these extended capabilities by inheriting from the class TS_Manager instead of TS_Controller.

Following example from the multicast application illustrates this facility: to allow the user to set up multicast connections interactively, a proxy client was implemented as TeleSoft controller. The user connects directly to the controller using a application such as telnet and is then able to set up or remove multicast sessions through simple commands.

8.9.3 Direct, platform dependent interaction

Sometimes it is necessary to break with the concept of platform independent controllers in order to be able to solve platform specific tasks.

The multicast protocol was first implemented as self-contained TeleSoft application. Point-to-multi point connections were being set up on a virtual ATM network and removed later on. Clients were modeled by traffic generators randomly creating and joining sessions while a 3D interface visualized the state of the network. Eventually, using TeleSoft, the application did not only run on the Simulation Kernel but also as a CORBA application on distributed network of workstations. The final goal however was to actually establish those connections on an ATM network, not only running a protocol emulation on a virtual one. Therefore it was necessary to replace some of the controllers by others which were capable of controlling ATM switches. Naturally, those controllers can only be utilized on a platform providing access to ATM switches, not in i.e. a pure simulation environment. These controllers can be considered a platform specific solution to a platform specific problem (Figure 62, part C, see the Section on the multicast application for details.)

In order to allow running such an application on different platforms, each system specific task should be encapsulated in one controller. To keep the application portable, it is sufficient to provide one implementation of those controllers for each platform only.

8.9.4 Mcast, an Example TeleSoft Application

The first application which has been developed using the development approach introduced by Rolf Stadler et al. is *mcast*, a multicast (point-to-multi point) service for broadband ATM networks (see [AUR97].) mcast does not only emulate networks, it also successfully controls an ATM network in our lab. The hardware is accessed through a software abstraction provided by XBind, a broadband kernel for devising service creation, deployment and management on ATM based multimedia networks ([AUR97a], [CHA96] and [LAZ94].)

The first implementation of mcast was done on a simulation system. In 1997, after a first prototype of TeleSoft was implemented, mcast was ported within a few weeks to the new platform. This extremely fast port was possible because of mcast's modular design which already based on the controller approach.

The final step was achieved by replacing mcast's virtual network by a real ATM network. Xbind with its device abstraction provided a simple and powerful interface for accessing switches through a CORBA interface. This allowed us to run the previously purely emulated multicast service after another few weeks on a real ATM network. This clearly domesticates the power and the feasibility of the TeleSoft approach. It is important to note that the same code is still used for simulation. The interface between the multicast service and XBind used for controlling the switches is encapsulated in the NodeServer, a TeleSoft controller which acts as a proxy for the ATM switch. What determines whether the mcast is an emulation only or real is which of the two implementation of the NodeServer are used, a dummy implementation or one which actually accesses the hardware.

The complete mcast application is very modular. There are four main components which may be used in almost any combination.

- A set of TeleSoft controllers implementing the multicast service, the link admission controllers and virtual switches.
- Xbind providing a CORBA interface for the hardware. This interface is used for two purposes: (A) to establish connections on the network and (B) to actually transport data through the connections once they're established.

- An external 3D interface used for monitoring the state of the network and the multicast service and controlling its parameters.
- A java based application which allows to set up multicast connections using the TeleSoft mcast application on the one side. It also initiated a MPEG video data stream which was sent over the network once a multicast video session was established. The interface and implementation of the transport protocol and other necessary layers is provided by XBind.

Figure 63 illustrates the interaction between each of the components. The top layer consists of the mcast application controllers, the middle layer contains the XBind device abstractions and the bottom layer the hardware. TeleSoft itself allows for two kinds of interaction. Via the visualizing interface (A) which uses the facility for monitoring and controlling and (B) by a special user interface which allows the user to interactively create and remove multicast sessions. The mcast nodeserver controllers establish the link to the XBind device abstractions. They enable mcast to access the ATM switches in order to establish transport connections. The transport itself is not part of the TeleSoft application but uses servers and tools provided by XBind. Once a connection is set up, a MPEG data stream is sent from the source (i.e. a VCR) through the network to the destination where it eventually is displayed on a screen (C). Signalling (connection setup) on one side and transport on the other are, though both based on XBind, completely separated.

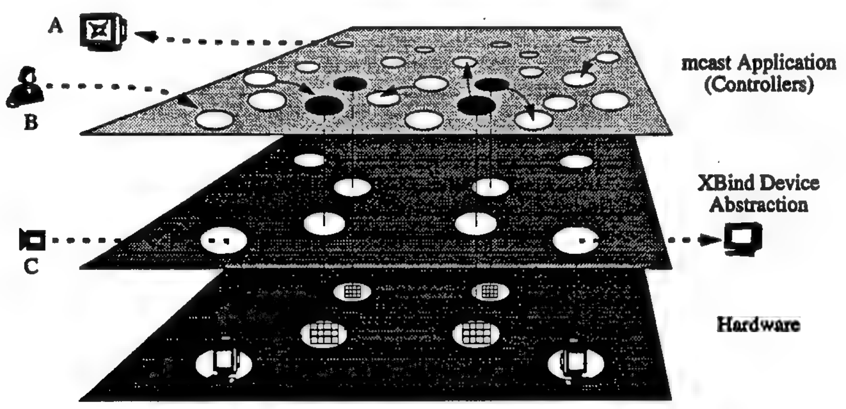


Figure 63 The mcast application: a multicast service for broadband ATM networks. The state of the TeleSoft controllers is visualized on a 3D interface (A) while the user has the possibility for interaction by using the same interface or the application proxy (B.) Once a session is established, MPEG video data was sent trough the network using the XBind transport interface (C.)

Figure 64 shows the topology of the network. Note that only the left part is real, the right part is emulated only. End-nodes are labelled *Wn*, ATM switches *Sn*.

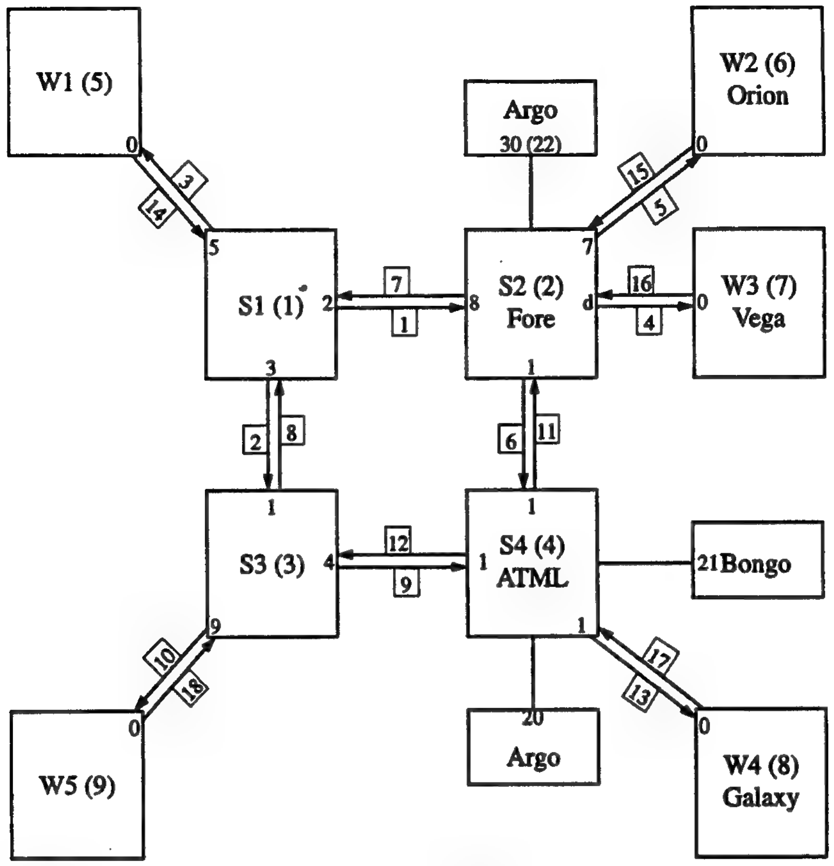


Figure 64 Sample mcast Network Topology

References

Introduction (from proposal)

[1] N.G. Aneroussis and A.A. Lazar, "Managing Virtual Paths on XUNET III: Architecture, Experimental Platform and Performance", Proceedings of *Fourth International Symposium on Integrated Network Management*, Santa Barbara, CA, May 1-5, 1995, pp. 370-384.

[2] N.G. Aneroussis and A.A. Lazar and D.E. Pendarakis, "Taming XUNET III", *ACM Computer Communications Review*, 1995, to appear.

[3] C. Aurrecoechea, M.C. Chan, A.A. Lazar, K.S. Lim and F. Marconcini, "Binding Model: Motivation and Description", *CTR Technical Report* 411-95-17, Center for Telecommunications Research, Columbia University, New York, June 1995.

[4] C. Aurrecoechea, M.C. Chan, A.A. Lazar, K.S. Lim and F. Marconcini, "xbind Version 1.0: A Corba Based Binding Architecture", *CTR Technical Report* 412-95-18, Center for Telecommunications Research, Columbia University, New York, June 1995.

[5] C. Aurrecoechea, A. Campbell, L. Hauw and H. Hadama, "A Model for Multicast for the Binding Architecture", *CTR Technical Report* 413-95-19, Center for Telecommunications Research, Columbia University, New York, June 1995.

[6] M. C. Chan, G. Pacifici, and R. Stadler, "Objective-driven performance management for broadband networks," *CTR Technical Report* 404-95-10, Center for Telecommunications Research, Columbia University, New York, May 1995.

[7] M. C. Chan, G. Pacifici, and R. Stadler, "Managing real-time services in multimedia networks using dynamic visualization and high-level controls," *Proceedings of the Third ACM International Conference On Multimedia*, (San Francisco, CA), November 1995

[8] The Common Object Request Broker: Architecture and Specification, Revision 1.2, published by the Object Management Group (OMG) and X/Open in Dec. 93

[9] HP, IBM, and SunSoft, "Multimedia system services architecture," June 1993.

[10] IMA Association, "Request for technology: Multimedia system services version 2.0," 1992.

[11] ITU-T Recommendation I.211; "Integrated Services Digital Network (ISDN), General Structure and Service Capabilities, B-ISDN Service Aspects"; 1991

[12] ITU, Open Sytem Interconnection -- *State Management Function* - ITU-T Rec.X.731, 1993

[13] Joint Object Services Submission, OMG TC Document 93.2.1

[14] A.A. Lazar, "A research agenda for multimedia networking," in *Workshop on Fundamentals and Perspectives on Multimedia Systems*, (Dagstuhl Castle, Germany), International Conference Center for Computer Science, July 1994.

[15] A.A. Lazar, "Challenges in multimedia networking," in *Proceedings of the International Hi-Tech Forum, Osaka'94*, (Osaka, Japan), pp. 24–33, February 1994.

[16] A.A. Lazar, S.K. Bhonsle and K.S. Lim, "A binding architecture for multimedia networks," *Journal of Parallel and Distributed Computing*, November 1995, to appear; also in the *Proceedings of the International Workshop on Multimedia Transport and Teleservices*, (Vienna, Austria), pp. 24–33, November 1994.

[17] A.A. Lazar, S.K. Bhonsle and K.S. Lim, "Implementing ATM Forum UNI over the Binding Architecture", working document, July 1994

[18] A.A. Lazar, G. Pacifici, and R. Stadler, ``An end-to-end connection service architecture for broadband networks,'' in *ICC'95 National Information Infrastructure Software Workshop*, (Seattle, WA), June 1995.

[19] G. Pacifici, and R. Stadler, "An architecture for performance management of multimedia networks," in *IFIP/ IEEE International Symposium on Integrated Network Management*, (Santa Barbara, California), pp. 74-186, May 1995.

[20] G. Pacifici, and R. Stadler, "Integrating resource control and performance management in multimedia networks," in *Proceedings of the IEEE International Conference on Communications*, (Seattle, WA), pp. 1541-1545, June 1995.

Rest of Report

[ATMF95] ATM Forum. *UNI V4: User to Network Interface specification*. ATM Forum, 1995.

[ADA97] C. Adam, A. A. Lazar, and M. Nandikesan, "Quality of Service Extensions to GSMP," Opensig'97, Cambridge, England, April 1997. http://comet.columbia.edu/publications/standard_contributions/qGSMP.ps.gz

[AUR97] Aurrecoechea C., Borla M., Stadler R..; "mcast: design of the service and its management", CTR Technical Report # 484-97-18; Columbia University New York, USA, December 1997.

[AUR97a] Aurrecoechea, C., Lazar, A.A. and Stadler, R., "Towards Building Manageable Multimedia Network Services", IFIP/IEEE International Conference on Management of Multimedia Networks and Services, Montreal, Canada, July 8-10, 1997.

[BHO97] Bhoj, P., Caswell, D., Chutani, S., Gopal, G. and Kosarchyn, M. *Management of new federated services*. IFIP/IEEE International Symposium on Integrated Network Management (IM'97). San Diego, CA, May 1997.

[BIE98] Bieri L.; "Measuring and Modeling the Performance of Software Controllers in Telecom Systems", Diploma Thesis; COMET Group, Columbia University New York, 1998.

[BRA97] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "RSVP Version 1 Functional Specification," Internet RFC 2205, September 1997.

[CHA97] Chan M.C., Pacifici G., Stadler R.; "Managing Multimedia Network Services", in Journal of Network and Systems Management, Vol. 5, No. 3, 1997.

[CHA97a] Chan M. C., Lazar A. A., Stadler R.; "Customer Management and Control of Broadband VPN services", IFIP/IEEE International Symposium on Integrated Network Management (IM '97), (San Diego, California), May 1997.

[CHA97b] Chan M. C., Hadama H., Stadler R.; "An architecture for broadband virtual networks under customer control", IEEE Network Operations and Management Symposium, (Kyoto, Japan), April 1996.

[CHA96] Chan M.C., Huard J.-F., Lazar A.A., Lim K.-S.; "On Realizing a Broadband Kernel for Multimedia Networks", In Proceedings of the Third COST 237 Workshop on Multimedia Telecommunications and Applications, Barcelona, Spain, November 25-27, 1996.

[CHA96a] Chan M.C., Pacifici G., Stadler R.; "Prototyping Network Architectures on a Super-computer"; Fifth International Symposium on High Performance Distributed Computing (HPCD-5), Syracuse, NY, USA, August 1996.

[CHA96b] Chan, M.C., Pacifici, G. and Stadler, R. *Realizing global control in multimedia networks.* IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '96), L'Aquila. Italy, October 1996.

[CHA95] Chan M.C., Pacifici G., Stadler R.; "Real-Time Emulation and Visualization of large Multimedia Networks", in Proceedings of the ACM Multimedia, Demonstrations Program; San Francisco, CA, USA, November 1995.

[CRA98] E. Crawly et al, "A Framework for QoS Routing in the Internet," work in progress, Internet Draft, April 1998.

[EUR97] EURESCOM P610 participants. *Providing framework, architecture and methodology for multimedia services management.* IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '97), Australia, October 1997.

[FUJ90] Fujimoto R.; "Parallel discrete event simulation", Communications of the ACM, October 1990, Vol. 33, No. 10, pp. 31-53.

[GAM95] Gamma E., Helm R., Johnson R., Vlissides J.; "Design Patterns, Elements of Reusable Object-Oriented Software", 1st Edition, Addison-Wesley, 1995.

[GUE98] R. Guerin, S. Kamat, A. Orda, T. Przygienda, and D. Williams, "QoS Routing Mechanisms and OSPF Extensions," work in progress, Internet Draft, January 1998.

[HYM91] Hyman, J.M., Lazar, A.A. and Pacifici, G. *Real-Time scheduling with quality of service constraints.* IEEE Journal on Selected Areas in Communications, September 1991.

[IONA96] Iona Technologies Ltd. *The Orbix architecture.* http://www.iona.ie/Orbix/arch/index.html, November 1996.

[ION97] IONA Technologies PLC; "ORBIX Programming Guide" and "ORBIX Reference Guide", http://www.iona.com/products/orbix/orbix/index.html.

[ITU92] ITU-T Recommendation M.3010 (1992). *Principles for a Telecommunications Management Network.* ITU, 1992.

[LAZ92] A. A. Lazar, "A Real-time Control, Management and Information Transport Architecture for Broadband Networks," in Proc. Int'l Zurich Seminar on Digital Communications, Zurich, Switzerland, March 1992.

[LAZ94] Lazar A.A., Bhonsle S. Lim K.S.; "A Binding Architecture for Multimedia Networks", Journal of Parallel and Distributed Systems, Vol. 30, Number 2, November 1995, pp. 204-216. Also in the Proceedings of the Multimedia Transport and Teleservices, Vienna, Austria, November 14-15, 1994.

[LAZ96] Lazar, A.A., Lim, K.S and Marconcini, F., "Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture," Journal of Selected Areas in Communications, Vol.14, No.7, September 1996, pp. 1214-1227.

[LAZ97] Lazar A.A.; "Programming Telecommunication Networks", IEEE Network, September/October 1997, (pp.8-18).

[LAZ98] A. A. Lazar, M. Nandikesan, and C. Shim, "Online Estimation of the Schedulable Region," Technical Report, Center for Telecommunications Research, Columbia University, New York, May 1998.

[NEW96] Newman, P., Hinden, R., Hoffman, E., Liaw, F.C., Lyon, T. and Minshall, G. *General Switch Management Protocol Specification. Version 1.0.* Ipsilon Networks, Inc., February 1996.

[OMG96] Object Management Group (OMG); "CORBA 2.0/IIOP Specification", Technical Document ptc/96-08-04; August 1996.

[PUT97] Puttkammer R.; "A Real-Time Simulator for Prototyping Telecommunications Applications", Diploma Thesis; COMET Group, Columbia University New York, 1997.

[PUT97a] Puttkammer R., Stadler R.; "Simulation Kernel 2.0 - Users Manual", Center for Telecommunications Research, Columbia University; March 1997.

[PNN96] "PNNI Vesion 1.0", ATM Forum, March 1996.

[ROS95] K. W. Ross, "Multiservice Loss Models for Broadband Telecommunication Networks," Springer-Verlag, 1995.

[RUS95] T. Russell, "Signaling System #7," McGraw-Hill, New York, 1995.

[SCH96] Schade, A., Trommler, P. and Kaiserswerth. *Object Instrumentation for Distributed Applications Management.* Distributed Platforms, Chapman & Hall, London, 1996.

[TINA95] Nilsson, G., Dupuy, F. and Chapman, M. *An overview of the Telecommunications Information Networking Architecture.* TINA 95, Melbourne, Australia, February 1995.

[TINA-NRA] TINA Network Resource Architecture Version: 3.0, Date: 10 February 1997. http://www.tinac.com/97/nra_v3.0_public.ps

[TINA-SA] TINA Service Architecture Version: 5.0, Date: 18 June 1997. http://www.tinac.com/97/sa50-main.ps

[ZHA97] Zhang, Sanchez, Salkewicz, Crawly, "Quality of Service Extension to OSPF," work in progress, Internet Draft, September 1997.

[ZHA93] Zhang, L., Deering, S., Estrin, D., Shenker, S. and Zappala, D. *RSVP: A new resource reservation protocol.* IEEE Network, September 1993.

9 Appendix: the Binding Interface Base

The Binding Interface Base is a collection of abstract CORBA IDL interfaces for representing multimedia resources in a network. The purpose of defining the BIB is to specify a standard set of interfaces to allow multimedia networking applications and systems based on CORBA to inter-work at the code level.

It is important to note that interfaces in the BIB represent only resources with local state information. By this we mean that the states of these resources have only local scope. For example, a resource such as a camera has only state information (e.g. size of its digitizing window, frame capture rate) and would qualify as an interface in the BIB. On the other hand, a connection management entity, by definition of its functionality would not, since its state space is composed of the set of distributed states of all switches and adapters whose connections it is managing.

This specification does not imply any structure about how these interfaces may be implemented or what language should be used. Thus, although some interfaces may directly be mapped into a single class in a specific language, for example, C++, this need not be true in general. The mapping of interfaces to their implementation is a design issue that is beyond the scope of this document.

Binding interfaces are organized into two categories :

1. *Data structures* that represent the basic constructs, types and constants used by the abstract interfaces.
2. *Abstract interfaces* that represent the multimedia resources organized into a interface hierarchy tree.

9.1 Data Structures

The following types of structures are defined in the specification :

BIBException. Exceptions allows error conditions resulting from the invocation of a call to be returned and propagated back to the caller. In a series of nested invocations, the `BIBException` structure allows information related to the originating exception as well as subsequent resulting exceptions to be consolidated and passed back up the calling stack.

EndPoint. `Endpoint`(s) describe transport and flow terminations in the network. Two endpoint structures have been defined - ATMEndpointId for a VP/VC termination and IPEndPoint Id for IP-address and port termination.

QOS. The structures for QOS are meant to specify the desired quality of service and profile for each level in the information transport stack. They will typically be used for connection management, monitoring and renegotiation (control).

9.1.1 Exceptions

The `bib_exception.idl` interface specifies a set of structures that shall be used to build and propagate exceptions. The exception types and the exception construction and propagation methods are defined in the `BIBExceptionHandler` C++ helper class.

As the root `VirtualResource` implementation class inherits from `BIBExceptionHandler`, all the BIB interfaces have capabilities to build and interpret BIB exceptions. Furthermore, BIB interface implementations can extend and customize the `BIBExceptionHandler` class functionality by inheriting from it. An example of useful customization that can be specific to each class is exception handling.

```
struct BIBExceptionInfo {
  string reason;
  string data;
  short type;
  short severity;
};
typedef sequence<BIBExceptionInfo> BIBExceptionInfoSeq;
struct BIBException {
  BIBExceptionInfoSeq bib_exception_info;
  string originating_interface;
  long originating_timestamp;
  string originating_objref;
};
```

```
exception Reject { BIBException bib_except; };
```

Structures

The `BIBExceptionInfo` structure shall be used when an interface generates an exception description. The signification of the structure fields is:

- `reason` - string that gives an exception description.
- `data` - additional data that allow to retrieve the internal state of the interface before the exception was raised.
- `type` - allows to classify an exception into a distinct category.
- `severity` - field used in exception handling. The severity level indicates what action should be taken at runtime after the exception occurence.

Each exception is described by a `BIBExceptionInfo` structure that contains the following:

- `bib_exception_info` - concatenates all the exception descriptions generated by the IDL interfaces, starting from the originating interface, and ending with the current interface.
- `originating_interface` - name of the interface that originated the exception
- `originating_timestamp` - the time when the exception occured in the originating interface
- `originating_objref` - stringified description that returns a reference to the BIB interface object that originated the exception when the method `CORBA::string_to_object()` is applied to it.

```
exception Reject { BIBException bib_except; };
```

All the BIB interfaces raise a single exception, called `Reject`. Any exception contains a `BIBException` structure, whose fields contain a full description of the exception.

Exceptions

- `BIBEx_Success` - the method execution was successful.
- `BIBEx_InvalidArgument` - invalid parameter value(s).
- `BIBEx_UninitializedArgument` - attempt to use the value of an uninitialized parameter.
- `BIBEx_ZeroCardinality` - the size of an array is zero
- `BIBEx_InvalidSize` - the size of an array is invalid

- BIBEx_UnsupportedOption - request for an option that is not supported in the current configuration of the system
- BIBEx_IncompatibleValueSet - the set of values does not comply with the constraints specific to the system.
- BIBEx_HardwareSpecific - exception originated at the hardware level.
- BIBEx_SoftwareSpecific - ran out of disk space or available memory, memory segmentation fault, core dump, etc.
- BIBEx_CORBASpecific - CORBA system exception
- BIBEx_ApplicationSpecific - exception originated at the application level.
- BIBEx_NotFound - attempt to get a reference to an inexistent object.
- BIBEx_InvalidAny - the structure extracted from a parameter of type any does not match the expected type.
- BIBEx_InvalidAuthentication - the authentication key used in an attempt to modify or release a capacity does not map the capacity server's key.
- BIBEx_CapacityExceeded - the request cannnot be executed because there is not enough capacity available.
- BIBEx_QOSViolation - the QOS constraints specified have been violated
- BIBEx_Unknown - unspecified exception uncovered by other failure codes

9.1.2 Endpoints

Endpoint(s) describe transport and flow terminations in the network. Two endpoint structures have been defined - ATMEndpointId for a VP/VC termination and IPEndPoint Id for IP-address and port termination.

```
enum EndPointFormat { IP_epf, ATM_epf};
struct ATMEndPointId {
  short port;
  short VPI;
  long VCI;
};
struct IPEndPointId {
  char ipadd[16];
  short port;
};
union EndPointId switch(EndPointFormat) {
  case IP_epf : IPEndPointId ip;
  case ATM_epf : ATMEndPointId atm;
};
struct EndPoint {
  EndPointFormat epformat;
  any data;
};
#endif
```

- epformat: The transport endpoint type encapsulated in the structure.
- `data`: The encoding of a particular endpoint structure.

9.1.3 QoS, Traffic Class, Link State

Here we list some of the basic data structures corresponding to basic network definitions such as network traffic descriptor, network QoS, link state, schedulable region, etc. These definitions will be used mostly by the BIB and the routing objects.

```
enum NetworkTrafficDescriptor {
    ANYTHING_ntd, VOICE_ntd, VIDEO_ntd, AUDIO_ntd, DATA_ntd
};
struct NetworkTrafficDescription {
    long peak_rate;   // kbps
    long ave_rate;    // kbps
    NetworkTrafficDescriptor type;
};

struct NetworkQOSProfile {
    long  max_delay;  // usec
    long  ave_delay;  // usec
    float loss;
    float ave_gap_loss;
    long  meas_era;   // usec
};

typedef unsigned long               IPAddress;
typedef sequence<long>              LongSequence;
typedef sequence<float>             FloatSequence;
typedef sequence<IPAddress>         IPAddressSequence;
typedef sequence<LongSequence>      LongDoubleSequence;
typedef sequence<FloatSequence>     FloatDoubleSequence;

typedef NetworkTrafficDescription TrafficDescription;
typedef NetworkQOSProfile         QOS;
typedef sequence<QOS>             QOSSequence;

struct TrafficClass{
  TrafficDescription traffic;
  QOS                qos_constraints;
};
typedef sequence<TrafficClass>      TrafficClassSequence;


struct Link{
  IPAddress src;
  short     src_outport;
  IPAddress dest;
  short     dest_inport;
  short     src_cr_id;          // 0 for physical links
};
typedef sequence<Link> LinkSequence;

struct LinkState{
  LongSequence          op;      // operating point
  FloatSequence         lambda; // call arrival rates (calls/s)
  FloatSequence         mu;      // call departure rates (calls/s)
};

struct CallLoad{
  FloatSequence         lambda; // call arrival rates (calls/s)
  FloatSequence         mu;      // call departure rates (calls/s)
  float                 ave_bw; // Average bandwidth for class
                                              of type ANYTHING_ntd
};

typedef LongDoubleSequence  CapacityPlaneList;

struct LinkTCDescription{
  Link                 link;
  TrafficClassSequence tc;
};

struct LinkSRDescription{
  Link              link;
  CapacityPlaneList sr;
};

struct LinkStateDescription{
  Link         link;
  LinkState    state;
};

struct SwitchCapacityDescription{
  IPAddress      switch_addr;
  unsigned long  switching_table_size;
};

struct CrossConnect{
  IPAddress node;
  short     inport;
  short     outport;
  short     outport_traffic_class_id;
  short     outport_cr_id;          // 0 for physical links
};
typedef sequence<CrossConnect> Route;
typedef sequence<Route>        RouteSequence;

struct RouteAndProbability{
  Route  route;
  float  probability;
};
typedef sequence<RouteAndProbability> RouteDistribution;

enum RoutingAlgorithmType{
SHORTEST_PATH, LOAD_BALANCING, DCR };
```

```
typedef sequence<RoutingAlgorithmType> RoutingAlgorithmTypeSequence;
```

The enumeration `NetworkTrafficDescriptor` gives the set of possible traffic characterizations defined at present. Further entries may be added in the future. The structure `NetworkTrafficDescription` (which is aliased as `TrafficDescription`) describes a set of traffic characteristics. Along with the associated QOS given by the structure `NetworkQOSProfile` (which is aliased as `QOS`), this describes a `TrafficClass`.

All `Links` are unidirectional point-to-point. The element `Link::src_cr_id` identifies contract region of the underlying link if the link in concern is not a physical link. When links are grouped together into a `LinkSequence`, they should be "breadth-first-search" ordered. The `LinkState` refers to statistics computed over a period specified external to the structure. The structures `LinkTCDescription`, `LinkSRDescription`, `LinkStateDescription`, and `SwitchCapacityDescription` are used by the event service to distribute state information.

SchedulableRegions are represented as the union of hyperplanes. An n-dimensional schedulable region S will be represented using an m x n non-negative matrix $A = (a_{ij})$, where m is a positive integer:

$$S = \bigcup_{i=1}^{m} H_i$$

where

$$H_i = \left\{ x \in N^n \middle| \sum_{j=1}^{n} \frac{x_j}{a_{ij}} \le 1 \right\}$$

The convention 0/0 = 0 and $x/0$ = infinity for $x > 0$ shall be adopted. The rows of the matrix A represent m (n-1)-dimensional hyperplanes; H_i is the set bounded by the i-th hyperplane and the coordinate hyperplanes. Schedulable regions are represented in IDL by the type `CapacityPlaneList`. A schedulable region `sr` holds a_{ij} in `sr[i][j]`.

A `CrossConnect` is defined as the connection from an input port to an output port of a switch. The element `CrossConnect::outport_cr_id` identifies contract region of the underlying link if the output link in concern is not a physical link. `Routes` are represented as sequences of `CrossConnects`. The structure `RouteDistribution` places a probability distribution on a set of routes used for load balancing.

9.2 Abstract Interfaces

The interface inheritance hierarchy of the BIB is presented in Figure 65.

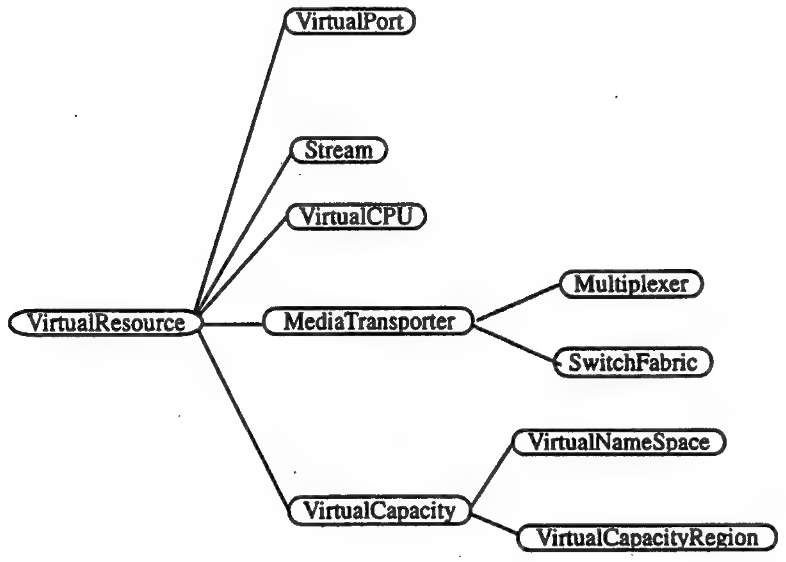


Figure 65 BIB Interface Inheritance Tree

9.2.1 The Virtual Resource

The *`VirtualResource`* interface is at the root of the BIB inheritance hierarchy. It contains basic attributes and methods inheritered by all BIB interfaces. These include methods for querying the version of the BIB, the instantiation time of an interface, the user and process identifier of the process implementing the interface and the actual interface name instantiated at runtime. The `VirtualResource` interface is derived from `CORBA::Object`.

```
interface VirtualResource
{
    readonly attribute string bibversion;
    readonly attribute long creation_time;
    readonly attribute string ip_addr;
    readonly attribute long pid;
    readonly attribute string userid;
    readonly attribute string interface_name;
    attribute string container;
    readonly attribute string xrm_plane;
    attribute string description;
};
```

Methods

The methods in this interface allow to set (at initialization time) and query the values of its attributes:

- `bibversion` - the version of Binding Interface Base that contains the interface IDL specification.
- `creation_time` - the instantiation time of the interface implementation.
- `ip_addr` - IP address of the machine on which the interface implementation runs
- `pid` - id of the process under which the interface implementation runs.
- `userid` - id of the user who launched the interface execution
- `interface_name` - interface name, as instantiated at runtime.
- `container` - stringified description of the object that contains this interface (its container). If the interface does not belong to a container, this variable shall be set to NULL. The method `CORBA::string_to_object()` applied to this attribute will return a reference to the CORBA container object.
- `xrm_plane` - the plane of the Extended Reference Model [...] to which the interface belongs conceptually.
- `description` - additional information about the interface.

Implementation Details

The implementation of a `VirtualResource` interface inherits from two classes: the `BIBExceptionHandler` and the `BIBContained` class.

`BIBExceptionHandler` integrates exception handling. It allows to access different fields of an exception structure, as well as to make and cascade exceptions. It is described in detail in section 1.1.5.

BIBContained is an abstract class that allows to set and get a pointer to the implementation of the container to which the resource belongs. Containers are described in detail in section 1.1.3.

9.2.2 The VirtualPort

The *VirtualTP* interface specifies the abstraction of a logical transport endpoint. Transport endpoints represent local termination points of an end-to-end network connection in the similar way ports model end points to a protocol stack in an operating system. The `VirtualTP` interface allows binding between transport protocols and network terminations. The interface includes methods for provisioning, QOS control and media transfer.

Methods

```
TPCId createGroup (in TPQOSSpec qos, in EndPoint ep)
       raises (InvalidQOSSpec, InvalidEndPoint, Reject);
```

The `createGroup` method is used to associate a multicast source endpoint (network termination) with a TPCId. We will refer to this identifier as the group identifier. The values of the members of the QOS parameters are to be used by the transport protocol engine for flow and rate control. Finally, the associated group identifier is returned by the method.

```
TPCId establish (in TPQOSSpec qos, in Endpoint ep)
       raises (InvalidQOSSpec, InvalidEndPoint, Reject);
```

The `establish` method is used to for both unicast and multicast to associate a network termination (specified by the endpoint parameter) with a TPCId. In the case of multicast, it is used by a sink for joining a group. In the case of a unicast, it is used by both ends to associate their network termination with a TPCId .

```
TPCId addLeaf (in TPCId grp, in TPQOSSpec qos,in Endpoint ep)
       raises (InvalidTPCid, InvalidQOSSpec, InvalidEndPoint,
       Reject);
```

The `addLeaf` method is used to inform the transport protocol that a sink (leaf) has joined the group. It specifies the sink requested QOS and network endpoint. Its main usage is for management, but it may also be use by the transport protocol engine for reliable multicast.

```
void release (in TPCId cid)
        raises (InvalidTPCid, TPReject);
```

The `release` method is used to dis-associate an endpoint with a transport connection identifier. The transport protocol engine should also close the appropriate device if necessary. When a group identifier is given as TPCId, the group associated with source is considered terminated. After a TPCId is released, methods using it will have an `InvalidTPCId` exception raised.

```
void setQOS (in TPCId cid, in TPQOSSpec qos)
        raises (InvalidTPCid, InvalidQOSSpec, TPReject);
```

The `setQOS` method is used to instruct the transport protocol engine of the requested QOS; that is, for a source, it specifies the traffic characteristics while, for a sink the QOS to be obtained from the network. The specified QOS will be monitored by the transport protocol engine at the sink and when sustain QOS violation occur, an event will be sent to an appropriate channel.

```
TPQOSSpec queryQOS (in TPCId cid, in TPQuery queryid)
        raises (InvalidTPCid, Reject);
```

The `queryQOS` method is used to query the transport protocol engine about the requested or obtained QOS of a specific connection (unicast or multicast).

9.2.3 The Stream

The *Stream* specifies an abstract interface of a media processing entity. Common examples of media processors include multimedia devices like cameras and microphones as well as software entities like codecs

To be specific, `Stream` specifies the stream control interface of a media producer consumer or translator. The interface includes methods for querying the list of multimedia stream formats and media types supported and generic control for pausing, resuming and querying the state of the stream.

```
#include "VirtualResource.idl"
```

```
#include "VirtualPort.idl"

// stream positioning
enum PositionOrigin { ABS_POS, REL_POS, MOD_POS };
enum PositionKey { BYTE_COUNT, SAMPLE_COUNT, MEDIA_TIME };

struct StreamPosition {
  PositionOrigin origin;
  PositionKey key;
  long value;
};

// Media Events
enum StreamState { ST_INIT, ST_PAUSED, ST_RUNNING, ST_KILLED };
enum MediaEventType { STREAM_INIT, STREAM_PAUSED, STREAM_RUNNING,
  STREAM_KILLED, STREAM_CHANGED_FORMAT, STREAM_CHANGED_POSITION,
  STREAM_CHANGED_SOURCE, STREAM_CHANGED_SINK };

struct MediaEventData {
  MediaEventType event_type;
  short stream_id;
};

typedef NameValueSeq StreamFormat;
typedef sequence<StreamFormat> StreamFormatList;

struct StreamInfo {
  StreamFormat format;
  long time_created;
  StreamState state;
};

typedef sequence<StreamInfo> StreamInfoList;
typedef sequence<NameValue> StreamCommand;

interface Stream : VirtualResource
{
  StreamInfo getInfo();
  boolean isOK();
  void changeFormat(in StreamFormat newformat)
       raises (Reject);
  StreamFormat getFormat() raises (Reject);
  void executeCommand(in StreamCommand command)
       raises (Reject);
  void resume () raises (Reject);
  void pause  () raises (Reject);
  void start  () raises (Reject);
  void stop   () raises (Reject);
  void attachPort(in VirtualPort port) raises(Reject);
  void detachPort(in VirtualPort port) raises(Reject);
};
```

Methods

```
void attachPort(in VirtualPort port) raises (Reject);
```

This method attaches a VirtualPort to the current device.

```
void detachPort(in VirtualPort port) raises (Reject);
```

id and a `Reject` exception is raised if the source is unable to remove the stream.

```
void pause() raises (Reject);
```

This method pauses a stream identified by the cid argument. An `InvalidIdentifier` exception is raised if the cid argument is not a valid transport connection id and a `Reject` exception is raised if the source is unable to pause the stream.

```
void resume() raises (Reject);
```

This method resumes a previously paused media stream specified by the `cid` argument. An `InvalidIdentifier` exception is raised if the cid argument is not a valid transport connection id and a `Reject` exception is raised if the source is unable to resume the stream.

9.2.4 The VirtualCPU

The *VirtualCPU* interface specifies the resource mangement and control interface of a media processor. A media processor can in general be responsible for generating or receiving multiple media streams. The VirtualCPU interface provides functionality for control of over how resources are allocated among these streams. In effect, the `VirtualCPU` interface allows control of the operating system scheduler in the arbitartion of computational resources among competing media proccessing tasks.

9.2.5 The Multiplexer

The *`Multiplexer`* interface models the functionality of the output ports of a physical switch. An output multiplexer consists of a set of buffers, a buffer manager, a scheduler and an estimator. The multiplexer components are used to mediate the contention among cells from different input links and among those of different traffic classes.

The interface methods allow to map traffic classes into the buffers, to change the scheduling and the buffer management policies and the available capacity estimation algorithm, as well as to retrieve these settings. They also provide capabilities to request and interpret traffic and QOS statistics.

Not all the switches support at the hardware level these QOS-related functionalities. A trivial configuration, consisting of a single traffic class mapped to a single buffer, a simple buffer manager, a First In First Out scheduler and an empty estimator is used to model an output multiplexer without QOS support.

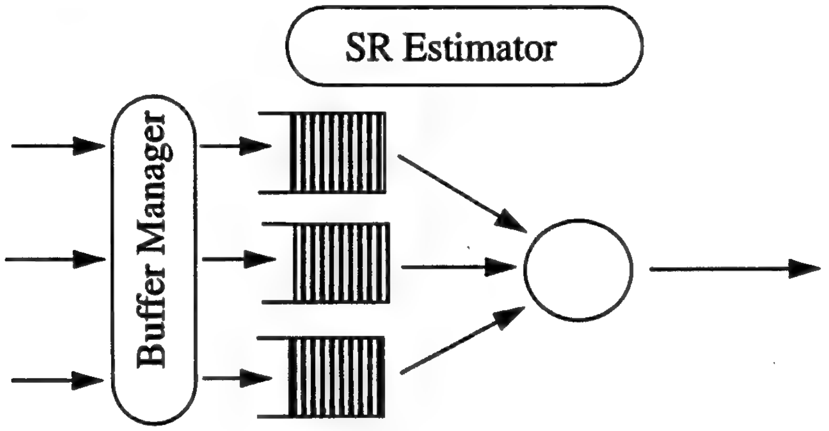


Figure 66 Modeling the Output Port of a Switch as Multiplexer

```
struct SchedulingPolicy {
  short classes;
  short type;
  ShortSeq parameters;
};

struct QueuePolicy {
  long queue_length;
  long thresh_hold;
  short action;
};
typedef sequence<QueuePolicy> QueuePolicyList;

struct BufferPolicy {
  short buffers;
  short classes;
  short type;
  QueuePolicyList queue_policies;
};

struct EstimatorDescriptor {
  short update_threshold;
```

```
    short type;
    any parameters;
  };

  interface Multiplexer : MediaTransporter
  {
    void setNbClassesBuffers(in short buffers,
                             in short classes,
                             in short classToBufferMap)    raises (Reject);

    void getNbClassesBuffers(out short buffers,
                             out short classes,
                             out short classToBufferMap)   raises (Reject);

    void setSchedulingPolicy(in SchedulingPolicy policy)raises (Reject);

    SchedulingPolicy getSchedulingPolicy()              raises (Reject);

    void setBufferMgmtPolicy(in BufferPolicy policy)    raises (Reject);
    BufferPolicy getBufferMgmtPolicy()              raises (Reject);

    void setEstimator(in EstimatorDescriptor estimator) raises (Reject);

    EstimatorDescriptor getEstimator()              raises (Reject);

    TrafficDescriptorList getTrafficMeasurements(in short reset,
                             out short classes,
                             inout ShortSeq quantitative,
                             inout ShortSeq measurements) raises(Reject);

    QOSConstraintsList getQOSMeasurements(in short reset,
                             out short classes,
                             inout ShortSeq classMask,
                             inout ShortSeq measurements) raises(Reject);

    void qosFailure(out short classes,
                             inout ShortSeq mask,
                             inout ShortSeq parameters)raises (Reject);
  };
```

Exceptions

- BIBEx_InvalidArgument - raised if either the type or the values in the Scheduling-Policy, BufferPolicy structures, or the estimator parameters are inconsistent.
- `BIBEx_HardwareSpecific` - raised by any method if the switch hardware does not support qGSMP or another equivalent QOS protocol and a request for a non-trivial buffer management, scheduling or schedulable region estimation setup is made. It can also be raised if the switch supports qGSMP, but does not support the specified scheduling, buffer management policy, or estimation algorithm requested.
- `BIBEx_QOSViolation` - raised when the QOS measurements received from the switch indicate that a QOS constraint has been violated.

Methods

setNbClassesBuffers. Sets the number of buffers and traffic classes (given by the arguments `buffers` and `classes`), and the buffer to be used by each traffic class on a specific port. The association between traffic classes and buffers is determined by the bitmap parameter `classToBufferMap` as follows:

Bit 0 is always set to 1. If the next bit that is 1 is at position n, then traffic classes 0 to n-1 are mapped to buffer 0. If the next bit that is 1 is in position m, then traffic classes n to m-1 are mapped to buffer 2. and so on. If the last bit that is set to 1 is at position p, then traffic classes p and higher are mapped to buffer k, where k is the number of 1's in the mask. k should equal the value set in the field `buffers`.

getNbClassesBuffers. Returns the number of buffers and traffic classes , and the mapping between the buffers and the traffic classes.

setSchedulingPolicy. Sets on the multiplexer the scheduling policy specified by the argument `policy`. The policy variable carries the following information about the scheduling policy: the number of traffic classes that are being scheduled, the scheduler type, and the parameters for each scheduler. The length of the parameter field is not specified, as different schedulers may require a different number of parameters. (i.e. First In First Out or Static Priority Scheduling schedulers require no parameters, the Weighted Round Robin scheduler requires the weights for each buffer, the MARS scheduler requires the H, h_1, h_2, ..., h_n, parameters).

getSchedulingPolicy() . Returns the scheduling policy that has been set up on the multiplexer.

setBufferMgmtPolicy. Sets on the multiplexer the scheduling policy specified by the argument policy. The `policy` variable carries the following information about the buffer management policy: the number of buffers that are being used, the number of traffic classes associated with these buffers, the type of the buffer manager (simple, threshold or proprietary buffer manager) and the queue policy for each of the buffers. The queue policy contains the length of the queue, the threshold for the queue, and the action that shall be performed on the cells of the queue once the threshold is reached.

getBufferMgmtPolicy(). Returns the buffer management policy that has been set up on the multiplexer.

setEstimator. Sets on the multiplexer the schedulable region estimation algorithm specified by the argument `estimator`. The `estimator` variable carries the following information about the estimation algorithm: type of the algorithm, the update threshold (defines the relative variation of the schedulable region for which the switch will send an alert to its controller), and the traffic classes for which estimation will be performed.

getEstimator(). Returns the description of the algorithm that estimates the schedulable region associated with the multiplexer.

getTrafficMeasurements. Returns the traffic measurements on the multiplexer. If the reset flag is true, then, after returning the present measurement, the measurement statistics are reset. The reset values of all traffic measurements shall be zero. The traffic classes for which measurements are provided are specified by the argument `classes`. The bitmap `quantitative` returns the parameters for which statistics are provided. The array `measurements` contains the measured values.

getQOSMeasurements. Returns the QOS constraints measurements on the multiplexer. If the reset flag is true, then, after returning the present measurement, the measurement statistics are reset. The reset values of all QOS constraints measurements shall be zero. The traffic classes for which measurements are provided are specified by the argument `classes`.The bitmap `classMask` returns the parameters for which statistics are provided. The array measurements contains the measured
values.

qosFailure. The hardware switch alerts the switch controller about a QOS violation. Only the traffic class parameters for which the QOS constraints were violated are specified in the masks and transmitted. The `classes` parameter gives the number of traffic classes experiencing QOS violation.

9.2.6 The Switch Fabric

The *`SwitchFabric`* interface specifies the abstraction of a logical switching or routing device. Methods in the interface allow the association and disassociation of input and output endpoints representing the setup or teardown of an internal route through the device. Other methods allow querying the number of endpoints supported in the device and generic functions to manage them.

The interface provides the finest level of granularity: it allows to add, remove or move a single channel. It takes as parameters the identifiers of the network interface cards between which it establishes the channel and the the names that identify, on each network card, the ends of the channel.

A multicast connection is created by adding additional branches to an existing unicast connection. A multicast connection can be removed branch by branch, using the `removeChannel` method several times, or by deleting directly the root of the multicast tree (using the `remove-Multicast` method).

The `SwitchFabric` interface can be used to establish channels between interface cards connected to different networks. The content of the name identifier structure used by the switch fabric has to be the same as the structure used in the name space implementation classes, as it is specific to the kind of network to which each network interface card is connected. For example, the Figure 67 shows the use of a `SwitchFabric` interface and of a number of *`ATMNameSpace`* objects to model the VPI/VCI mapping functions of a physical ATM switch.

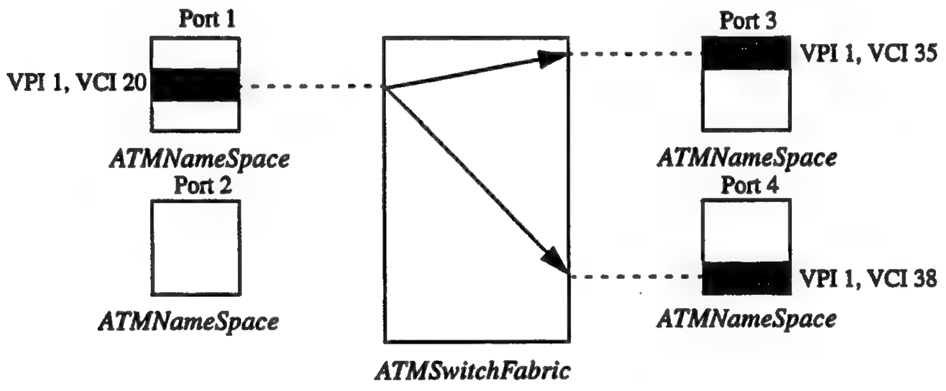


Figure 67 ATM Name Space Mapping Using an ATMSwitchFabric

```
interface SwitchFabric : MediaTransporter
{
  short addChannel(in short in_port, in NameIdentifier in_entry,
                       in short out_port,
                       in NameIdentifier out_entry,
                       in any connection_info)          raises (Reject);

  short removeChannel(in short in_port,
                       in NameIdentifier in_entry,
                       in short out_port,
                             in NameIdentifier out_entry,
                             in any connection_info)        raises (Reject);

    short removeMulticast(in short in_port,
                             in NameIdentifier in_entry,
                             in any connection_info)        raises (Reject);

    short moveChannel(in short in_port,
                             in NameIdentifier in_entry,
                             in short old_out_port,
                             in NameIdentifier old_out_entry,
                             in short new_out_port,
                             in NameIdentifier new_out_entry,
                             in any old_connection_info,
                             in any new_connection_info)      raises(Reject);

    void getOutputEntries(in short in_port,
                             in NameIdentifier in_entry,
                             out ShortSeq out_ports,
                             out NameIdentifierList out_entries) raises(Reject);

    void getInputEntry(out short in_port,
                             out NameIdentifier in_entry,
                             in short out_port,
                             in NameIdentifier out_entry)  raises (Reject);

    long getCapacity()                                      raises (Reject);
  };
```

Exceptions

- `BIBEx_InvalidArgument` - raised if the method parameters are out of the allowable ranges or in the methods `removeChannel` or `removeMulticast` if the specified connection does not exist.
- BIBEx_InvalidSize - raised if the sizes of the sequence parameters for a method are not the same.
- `BIBEx_HardwareSpecific` - raised in the `addChannel` or `removeMulticast` methods, when a general problem related to the manner in which multicast is supported by the switch is encoutered.
- `BIBEx_InvalidAny` - can be raised in methods that have parameters of the type `NameIdentifierList` or `ConnectionInfoList` type.

Methods

addChannel. Adds a channel (association) between two network interface cards. The cards are identified by the `in_port` and `out_port` parameters. The ends of the channel on each card are defined by the `in_entry` and `out_entry` parameters. Any additional information about the association is stored in the `connection_info` parameter.

removeChannel. Removes a channel (association) between two network interface cards. The cards are identified by the `in_port` and `out_port` parameters. The ends of the channel on each card are defined by the `in_entry` and `out_entry` parameters. Any additional information about the association is stored in the `connection_info` parameter.

removeMulticast. Removes the multicast connection rooted in the `in_port` network interface card and identified on that card by `in_entry`.

moveChannel. Changes an association in the table. The connection originating from the in_entry of the in_port network interface card is redirected from the old_out_entry index of the old_out_port network interface card to the new_out_entry index of the new_out_port network interface card. Any additional information about the association is updated with the content of the new_connection_info parameter, instead of the old_connection_info description.

getOutputEntries. Returns the lists of output ports and output name identifiers associated with the input port in_port and the name identifier in_entry. The *n*-th element of the out_entries list represents the end of the channel defined on the port located on the *n*-th position of the sequence out_ports. If there are no output entries, then the method returns two sequences of length zero.

getInputEntry. Returns the input port and the input identifier associated with channel ending on the output identifier out_entry of the port out_port.

9.2.7 The VirtualCapacity

The interfaces in this category specify generic capacity abstractions for multimedia resources. In general, two forms of resources exist in the network: *homogenous* resources, like bandwidth, and *heterogeneous* resources, like name space identifiers.

Homogenous resources of a given size are identical and thus resources of this form can be characterized by a capacity space. An example of a homogenous resource is bandwidth. It is characterized by some basic unit (*e.g.* bits/sec or number of calls that can be supported on a given link) and is a product of buffer management and scheduling mechanisms. If the capacity of a 100 Mbps physical link is defined in terms of peak rate, there is no difference between two 50 Mbps portions of the total capacity.

On the other hand, heterogeneous resources are individually unique. A name space resource is characterized by the number of distinct logical identifiers that can be defined in a naming domain and is a product of a given particular addressing scheme. For example, for an ATM network interface card, it is the set of all possible VP/VC combinations.

In the scope of the current BIB specification, only one form of heterogeneous resource is considered - namely name space identifiers. The capacity of a name space resource is abstracted via the `VirtualNameSpace` interface, while the capacity of a homogenous resource like bandwidth is abstracted via the `VirtualCapacityRegion` interface.

Capacities can be acquired by several (competing) service providers. For security and authentication purposes, each capacity object keeps track of its own partition between several external servers. Each time an external server acquires a capacity, it is assigned an authentication key. The key is encrypted using some algorithm and transmitted over the network. In order to release a capacity, the server needs to provide the key it was assigned at acquisition time.

```
interface VirtualCapacity : VirtualResource
{
  string getCapacityType()  raises (Reject);
  short getDimension()      raises (Reject);
};
```

getCapacityType(). Returns the capacity type (homogenous or heterogenous).

getDimension(). Returns the dimension of the capacity. For a homogeneous resource, (like `VirtualCapacityRegion`), this is the dimension of the capacity space (or the number of resource classes defined for it). For a heterogeneous resource (like `VirtualNameSpace`), this is the number of distinct parameters used to define a resource identifier.

9.2.8 The Virtual Name Space

The `VirtualNameSpace` is an abstract interface that specifies a set of methods to define and allocate identifiers from a hierarchically structured naming domain. The `NameIdentifier` type is of the CORBA `any` type. For each implementation class, a structure that indicates how to interpret the `any` type shall be specified.

The methods in this IDL interface allow an external service provider (*e.g.* a connection manager) to reserve or release name space resources. Reservation operations can be done for a single identifier, for a range of identifiers, or for identifiers matching a certain pattern.

A naming domain generally consists of a number of sub-domains each of which has a fixed length. Identifiers in a domain may consist of alphabets only, numbers only, or alpha numeric characters. This is specified by the `IdentifierType` field of the `NameSpaceRep` structure. A valid name in a domain consists of all combinations of characters (with the exception of wildcard characters) lexicographically greater or equal to a specified identifier: the `start_range` field of the `NameSpaceRep` structure. In this scheme, it is assumed that all names can be ordered lexicographically in the manner detailed by the standard ASCII character set.

The name identifier implementation structure members are arranged in the order of their significance. The most significant member is defined first, the least significant is defined last. For example, for an ATM name identifier, the structure members will be defined in the following order: Port, Virtual Path and Virtual Channel.

The length of a name identifier range is always specified in terms of the least significant member of the structure. For example, if an ATM virtual path contains *256* virtual channels, in order to reserve two consecutive virtual paths, one needs to acquire a range of *512 = 2 * 256* virtual channels.

Two kinds of wildcard characters are defined for pattern matching. The "*?*" character implies a single substitution of the wildcard with any legal character of the correct `IdentifierType`. The "*" character imples a Klene star substitution in the conventional sense.

```
typedef any NameIdentifier;
typedef sequence<NameIdentifier> NameIdentifierList;

enum IdentifierConstraint { CONSECUTIVE, NON_CONSECUTIVE };

struct NameSpaceRep {
  NameIdentifier start_range;
  short domain_len;
  IdentifierType type;
};
typedef sequence<NameSpaceRep> NameSpaceRepList;

interface VirtualNameSpace : VirtualCapacity
{
  boolean acquireSpecificIdentifier(in NameIdentifier name,
                        inout string authentication_key) raises (Reject);

  NameIdentifierList acquireIdentifiersByRange(in long num,
                        in NameIdentifier start_range,
                        in NameIdentifier end_range,
                        in IdentifierConstraint constraint,
                        inout string authentication_key) raises (Reject);
```

```
    NameIdentifierList acquireIdentifiersByPattern(in long num,
                           in NameIdentifier pattern,
                           in IdentifierConstraint constraint,
                           inout string authentication_key) raises (Reject);

    void releaseSpecificIdentifier(in NameIdentifier name,
                           in string authentication_key)    raises (Reject);

    void releaseIdentifiersByRange(in NameIdentifier start_range,
                           in NameIdentifier end_range,
                           in string authentication_key)    raises (Reject);

    void releaseIdentifiersByPattern(in NameIdentifier pattern,
                           in string authentication_key)    raises (Reject);

    void setNameSpace(in NameSpaceRepList name_space)       raises (Reject);

    NameSpaceRepList getNameSpace()                         raises (Reject);
};
```

Exceptions

- `BIBEx_InvalidArgument` - raised if methods contain arguments out of range.
- `BIBEx_InvalidSize` - raised in `acquireIdentifiersByRange` and `acquireIdentifiersByPattern` when the number of name identifiers requested is greater than the length of the available range.
- `BIBEx_InvalidAny` - can be raised in any method that has parameters of the type `NameIdentifier`. See section 1.1.5.3.
- BIBEx_CapacityExceeded - raised if the all the name space identifiers have been assigned.
- `BIBEx_InvalidAuthentication` - see section 1.1.5.3.

Methods

acquireSpecificIdentifier. Attempts to acquire the name identifier specified by the argument `name`. Returns `TRUE` if the attempt is successful and `FALSE` otherwise.

acquireIdentifiersByRange. Attempts to acquire `num` name identifiers in the range bounded by `start_range` and `end_range`, with the constraint specified in `constraint`. Returns the requested list of identifiers if the attempt is succesful, an empty list (of length zero) otherwise.

acquireIdentifiersByPattern. Attempts to acquire `num` numbers of `NameIdentifiers` using a template specified by the argument `pattern` with the constraint specified in `constraint`. Returns the requested list of identifiers if the attempt is succesful, an empty list (of length zero) otherwise.

releaseSpecificIdentifier. Releases the `NameIdentifier` specified by the `name` argument.

releaseIdentifiersByRange. Releases all the name identifiers that are contained in the range `[start_range,end_range]` and match the authentication key.

releaseIdentifiersByPattern. Releases all the name identifiers that match the template `pattern` and match the authentication key.

setNameSpace. Defines a name space using the argument `name_space`.

getNameSpace(). Returns the representation of the name space.

9.2.9 The Virtual Capacity Region

The `VirtualCapacityRegion` interface specifies an abstract capacity space and state of a homogenous resource. The capacity space is expressed in terms of the number and types of resource classes that the resource can support while guaranteeing a set of constraints.

The current amount of resource allocated can be represented by an *operating point*. If the operating point is inside the virtual capacity region, it is guaranteed that all the allocated resources will satisfy a given set of constraints.

This interface does not define the resource classes that define the axes of its space, nor does it specify what is the unit on each axis. This is done in the inheriting interfaces.

Instead of acquiring a single type of resource of a specific class, one can acquire an entire subset of the virtual capacity region. For example, if a VP is setup on an ATM switch, then the resources allocated to that VP will form a separate *Contract Region.*

The *MultimediaCapacityRegion* interface abstracts the computational capacity of a media processor. Its dimensionality is equal to the number of service classes supported by the media processor (e.g. MPEG I video, CD quality audio etc.).

The *SchedulableRegion* interface abstracts the multiplexing capacity of a switching device. Its dimensionality is equal to the number of traffic classes or distinct flow specifications supported by the device. A set of cell-level QOS constraints (maximum delay and loss probability bounds) are associated with each traffic class.

```
typedef sequence<LongSeq> CapacityPlaneList;
typedef sequence<LongSeq> CapacityPointList;

interface VirtualCapacityRegion : VirtualCapacity
{
  boolean isWithin(in LongSeq a_point)                            raises (Reject);

  boolean acquireCapacity(in short resource_class,
                          inout long size,
                          inout string authentication_key)  raises (Reject);

  void releaseCapacity(in short resource_class,
                          in long size,
                          in string authentication_key)     raises (Reject);

  boolean modifyCapacity(in short old_resource_class,
                          in long old_size,
                          in short new_resource_class,
                          inout long new_size,
                          inout string authentication_key) raises (Reject);

  boolean acquireCapacityRegion(inout CapacityPlane cap,
                          inout string authentication_key) raises (Reject);

  void releaseCapacityRegion(in CapacityPlane cap,
                          in string authentication_key)     raises (Reject);

  boolean modifyCapacityRegion(in CapacityPlane old_cap,
                          inout CapacityPlane new_cap,
                          inout string authentication_key) raises (Reject);

  void setCapacitybyPlanes(in CapacityPlaneList cap)          raises (Reject);

  void setCapacitybyPoint(in CapacityPointList cap)           raises (Reject);
};
#endif
```

Exceptions

- `BIBEx_InvalidSize` - raised if the size of an array that represents a point or a plane is not equal to the dimension of the virtual capacity region.
- `BIBEx_InvalidArgument` - raised if one of the resource class parameters is negative or greater than the number of virtual capacity region classes
- `BIBEx_InvalidAuthentication` - raised if the key provided to modify a capacity does not match the local key.
- BIBEx_CapacityExceeded - raised if there is not enough resource available left.

Methods

isWithin. Returns `TRUE` if the point `a_point` is inside the virtual capacity region, `FALSE` otherwise.

setCapacitybyPlanes. Initializes the capacity using as input a set of hyperplanes.

setCapacitybyPoint. Initializes the capacity using as input a set of points.

acquireCapacity. Acquires `size` units of a resource class. If the request is successful, the method returns `TRUE`. If there is insufficient capacity, the method returns `FALSE`, and the variable `size` will contain the amount of available resource.

releaseCapacity. Releases `size` units of a resource class.

modifyCapacity. Changes the type and/or the size of a resource class acquired. The operation is done atomically, *i.e.* it is guaranteed that the old capacity will not be released until the new one was acquired. If the request is successful, the method returns `TRUE`. Otherwise, it returns `FALSE`, and the `new_size` parameter will contain the available amount of the resource of type `new_resource_class`. If the acquisition is unsuccessful, the client that has requested the modification will still own the old capacity.

acquireCapacityRegion. . **releaseCapacityRegion.** . **modifyCapacityRegion.** These three methods allow to partition the virtual capacity region into several subsets.

9.2.10 The Schedulable Region

The *schedulable region (SR)* is the space of possible combinations of calls of each traffic class that an output link can handle while guaranteeing quality of service [...]. Figure 44 gives an illustration.

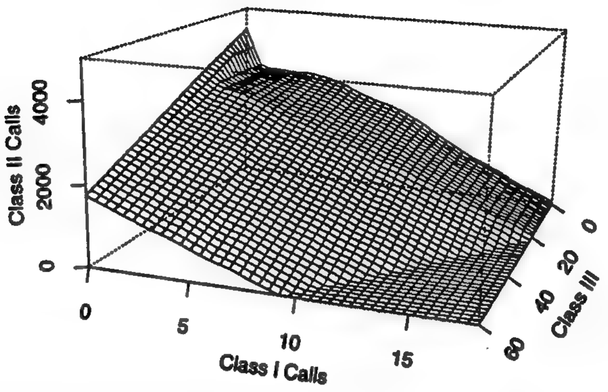


Figure 68 The region below the shaded surface is the schedulable region associated with a multiplexer supporting three traffic classes.

A *Traffic class* represents a statistical model for an information stream. It shall be defined by a peak rate and an additional stochastic characterization such as *video, voice, data*, or *peak_rate*. A traffic class that carries data is specified through its average rate. The peak_rate class includes the information streams that do not fit into any traffic class statistical model. Therefore, the statistical multiplexing gain cannot be exploited for this class, and the resources are allocated to it using peakrate allocation.

A set of cell-level QOS constraints (maximum delay and loss probability bounds) are associated with each traffic class.

```
enum StochasticDescriptor {Video,Audio,Data,PeakRate};

struct TrafficDescriptor {
  StochasticDescriptor sd;
  long base_peak_rate;
 long base_mean_rate;
};

struct QOSConstraints {
  short max_delay;
  short avg_delay;
  float prob_of_loss;
  short max_gap_loss;
};

typedef sequence<TrafficDescriptor> TrafficDescriptorList;
typedef sequence<QOSConstraints> QOSConstraintsList;

interface SchedulableRegion : VirtualCapacityRegion
{
  void setTrafficDescriptors(in short autoset,
                       in TrafficDescriptorList tdList)  raises (Reject);

  TrafficDescriptorList getTrafficDescriptors()         raises (Reject);

  void setQOSConstraints(in short autoset,
                       in QOSConstraintsList qoscList)   raises (Reject);

  QOSConstraintsList getQOSConstraints()                raises (Reject);
};
```

Exceptions

- `BIBEx_InvalidSize` - raised if the size of an array used to represent a point or a plane is not equal to the dimension of the schedulable region.
- `BIBEx_InvalidArgument` - raised if one of the resource class parameters is negative or greater than the number of schedulable region classes
- `BIBEx_HardwareSpecific` - raised in `setTrafficDescriptors` and `setQOSCon-strains` when the switch hardware cannot support the requested set of traffic descriptors, or the requested set of QOS constrains either because they are too stringent or because some of them cannot be measured on the switch, or if the hardware cannot support the requested set of traffic/QOS parameter types simultaneously (because they are too stringent).

Methods

setTrafficDescriptors. Sets the traffic descriptors (stochastic characterization and peakrate) for the schedulable region traffic classes. If QOS support is provided, the implementation shall also transmit the new settings to the ATM switch hardware, using the qGSMP protocol.

getTrafficDescriptors(). Retrieves the set of traffic descriptors defined for the schedulable region traffic classes.

setQOSConstraints. Sets the QOS constraints (maximum delay, probability of loss) for the schedulable region traffic classes. If QOS support is provided, the implementation shall also transmit the new settings to the ATM switch hardware, using the qGSMP protocol.

getQOSConstraints(). Retrieves the set of QOS constraints defined for the schedulable region traffic classes.

9.2.11 The Switch Mapping Capacity

The `MappingCapacity` interface Switch models the capacity of a switching device routing table. It is a uni-dimensional homogeneous capacity. The resource class unit is one connection (a mapping between an input and an output identifier). Due to the trivial definition of its single resource class, the interface does not define any additional methods or exceptions from those inherited from the `VirtualCapacityRegion` interface.

9.3 Other Core Xbind Interfaces

9.3.1 Event Channels

The `EventChannel` defines an interface to a CORBA event dispatching service. Servers which emit events "push" them to an event channel. Other servers which are interested to receive these events register their interest with the event channel. When an event is received by an event channel, it is relayed to the appropriate registered server. Events have a standard structure that contains important information fields pertaining to details that generated the event. A data field is also defined to allow custom data to be encapsulated within the event structure. Servers which want to receive events must inherit from the abstract interface `BIBEventHandler` and provide an implementation for the `eventCallback()` method.

```
struct BIBEvent {
  string event_name;
  BIBObjRef originating_interface;
  long timestamp;
  string originating_host_ip;
  any data;
};
struct OperationEventData {
  short operation_type;
  char principal[BIB_PRINCIPAL_LEN];
  char operation_name[BIB_OPERATION_NAME_LEN];
};
interface BIBEventHandler
{
  oneway void eventCallback(in BIBEvent event);
};

interface EventChannel
{
    void registerListener(in BIBObjRef emitter,
                          in string event_name)      raises (Reject);

    void unregisterListener(in BIBObjRef emitter,
                          in string event_name)      raises (Reject);

    oneway void pushEvent(in BIBEvent event);
};
```

Exceptions

- `BIBEx_InvalidArgument` - raised if the emitter argument of the `registerListener()` and `unregisterListener()` call is invalid.

Types

Each event is described by the `BIBEvent` structure, that contains the following fields :

- `event_name` - string representing the name of the event
- `originating_interface` - The name of the IDL interface which originated the event
- `timestamp` - The time which the event was generation
- `originating_host_ip` - The IP address of the host on which the server responsible for generating the event resides
- `data` - A general field for carrying event type specific data

The `OperationEventData` structure holds data structures specific to events generated by operation on attributes or method invocations. The structure is to fit into the data field of the `BIBEvent` structure. The fields of the `OperationEventData` structure are as follows:

- `operation_type` - type of the operation that generated the event. Valid types:
  - -0 - incoming method invocation
  - -1 - outgoing method invocation
  - -2 - incoming get attribute operation
  - -3 - outgoing get attribute operation
  - -4 - incoming set attribute operation
  - -5 - outgoing set attribute operation
- `principal` - stringified object pointer of the IDL interface that originated the event
- `operation_name` - name of the operation

The `BIBEventHandler` interface defines an abstract interface with a single pure abstract method `eventCallback()` which should be overridden by the deriving implementation in order to receive events.

Methods

registerListener. Registers an interested `BIBEventHandler` interface with the event channel. The `emitter` and `event_name` arguments specify the stringified object pointer of the interface and the name of the event that the listener is interested respectively.

unregisterListener . De-registers a listener from the event channel.

pushEvent. “Pushes” events into the event channel.

9.3.2 The Managed Controller

The `ManagedController` interface defines a simple set of generic management functions for general-purpose CORBA servers. These include capability to restrict access to the IDL interfaces implemented by objects within the server through a lock, temporarily block all incoming invocations to the server, monitor all incoming and outgoing invocations to and from the target server and finally terminate the execution of the server process. The granularity of operation of these functions are limited to per-server (or CORBA process). The typical usage of the `ManagedController` interface is through inheritance by interfaces implementing server functionality, for instance the `VirtualContainer` interface.

Monitoring of a `ManagedController` is achieved by a “listening” for event emissions generated by the controller in response to method invocations or attribute access. Each controller has a default event channel which it will “push” these events into. Exactly which types of invocations will result in an event emission will depend on the state of the controller.

```
enum executionState {
  CTRL_PAUSED, CTRL_INIT, CTRL_RUNNING, CTRL_DEFUNCT };
enum accessState {
  CTRL_READ_LOCKED, CTRL_UNLOCKED, CTRL_EXCL_LOCKED };
enum monitorState {
  CTRL_MONITOR_INCOMING, CTRL_MONITOR_OUTGOING,
  CTRL_MONITOR_ALL, CTRL_MONITOR_NONE };
struct controllerInfo {
  long time_started;
  executionState ex_state;
  accessState ac_state;
  monitorState mon_state;
};

interface ManagedController
{
  readonly attribute controllerInfo basic_state;
  string describeController();
  string getEventChannel();

// control functions
  void killController()                               raises (Reject);
  void setExecutionState(in executionState ex_state)  raises (Reject);
  void setAccessState(in accessState ac_state)        raises (Reject);
  void setMonitorState(in monitorState mon_state)     raises (Reject);

// convenience functions
  executionState getExecutionState()                  raises (Reject);
  accessState getAccessState()                        raises (Reject);
  monitorState getMonitorState()                      raises (Reject);
};
```

Types

```
enum executionState {
  CTRL_PAUSED, CTRL_INIT, CTRL_RUNNING, CTRL_DEFUNCT };
```

Lists all possible execution states of the controller. When a `ManagedController` interface is first instantiated, it is in the CTRL_INIT state.

When the implementation has completed its initialization and is ready to accept calls, it will change its state to `CTRL_RUNNING`. A controller in the `CTRL_PAUSED` state cannot receive any CORBA invocations. A controller in the `CTRL_DEFUNCT` state is assumed to be non-operational and cannot receive any further CORBA invocations either.

```
enum accessState {
  CTRL_READ_LOCKED, CTRL_UNLOCKED,CTRL_EXCL_LOCKED};
```

Lists all possible interface access state of the controller. By default, a controller is in the `CTRL_UNLOCKED` state. This means that there is no restriction in the invocation of its methods and attributes. A controller in the `CTRL_READ_LOCKED` state allows the user who locked it unrestricted access to its methods and attributes but all others only read-access to its attributes and no access to its methods. A controller in the `CTRL_EXCL_LOCKED` state allows the user who locked it unrestricted access but all others completely no access to its attributes or methods.

```
enum monitorState {
  CTRL_MONITOR_INCOMING, CTRL_MONITOR_OUTGOING,
  CTRL_MONITOR_ALL, CTRL_MONITOR_NONE };
```

Lists all possible monitoring capabilities of the controller. By default, the controller is in the `CTRL_MONITOR_NONE` state in which none of its methods or attributes are monitored. This means that calls to any method or attribute will not result in the emission of any events. When a controller is in the `CTRL_MONITOR_OUTGOING` state, all method invocations and attribute access made by the controller to any other IDL interface not in the controller's address space will trigger the generation an event. Similarly, when a controller is the `CTRL_MONITORING_INCOMING` state, any method invocation or attribute access to its interface will result in the generation of an event. Finally, when a controller is in the `CTRL_MONITOR_ALL` state, it generates events in response to any incoming or outgoing calls. Note that all events generated are forwarded to the default event channel.

Methods

basic_state; get/set the state of the controller, reflected in the `controllerInfo` structure as follows:

- `time_started` - the instantiation time of the controller
- `ex_state` - the current execution state of the controller
- `ac_state` - the current access state of the controller
- `mon_state` - the current monitoring state of the controller

describeController. Returns a string containing the description of the controller.

getEventChannel. Returns a stringified description of the current event channel object pointer.

setExecutionState . setAccessState. setMonitorState. Change, respectively, the execution, access and monitoring state of the controller.

getExecutionState. getAccessState. getMonitorState. Return, respectively, the current execution, access and monitoring state of the controller.

*U.S. GOVERNMENT PRINTING OFFICE: 1999-610-130-81109

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.